Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

ESTENSIONE A DUE STADI DI MODELLI VAE PER LA GENERAZIONE DI IMMAGINI

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
MATTEO TRENTIN

Sessione II
Anno Accademico 2018/2019

Introduzione

La disciplina nota come apprendimento automatico, o *machine learning*, rappresenta un importante argomento nell'informatica, sia dal punto di vista della ricerca, sia nell'industria; l'approccio più popolare a questo campo è sicuramente quello del *deep learning*, che utilizza le reti neurali come mezzo di approssimazione e rappresentazione di funzioni altrimenti troppo complesse da determinare e calibrare manualmente.

Una particolare applicazione di queste tecniche è quella della generazione di nuovi dati, come ad esempio contenuti audio o immagini; dal punto di vista della qualità dei risultati ottenuti, risulta essere molto efficace il modello delle GAN (*Generative Adversarial Network*, [8]). Un altro approccio, caratterizzato da una maggiore semplicità di calibrazione e dall'efficacia nel codificare le informazioni sotto forma di variabili latenti, è quello dei VAE (*Variational Autoencoder*, [13], [14]), illustrato in questa tesi.

Caratteristica fondamentale dei VAE è l'obiettivo di creare una rappresentazione latente del dataset utilizzato per la calibrazione (o training) della rete, tale per cui per ogni punto del dataset X esista una configurazione delle variabili latenti che porta il modello a generare qualcosa di simile ad X ([7]); si cerca inoltre di portare la distribuzione delle variabili latenti $Q(z|X)$ ad essere simile ad un *prior* $P(z)$.

In questa tesi viene analizzato un miglioramento ([5]) proposto all'architettura dei VAE, un'estensione a due stadi con l'obiettivo di migliorare la funzionalità generativa di questo tipo di modello; questo è composto da due VAE, dei quali il primo ha il compito di apprendere la rappresentazione latente dei dati di input, e il secondo ha il compito

di migliorare la rappresentazione latente del primo, utilizzandola come dato di training per portare ad una distribuzione più simile al prior; poiché in fase di generazione si effettua un campionamento su quest'ultimo, uno spazio latente simile ad esso avrà una probabilità più alta di contenere informazioni utili nei punti campionati.

Viene inoltre mostrata una possibile estensione condizionale dell'architettura a due stadi, sperimentata su due differenti dataset, con l'intento di produrre dati aventi caratteristiche a scelta, e non quindi generati del tutto casualmente.

Struttura della tesi

Nel Capitolo 1 viene effettuata una breve introduzione ad alcuni fondamenti del deep learning, utilizzati nel resto della tesi, nello specifico: reti neurali, reti neurali convoluzionali e training.

Nel Capitolo 2 vengono descritte due metriche per le prestazioni dei metodi generativi, ovvero Inception Score (IS)[18] e Fréchet Inception Distance (FID)[10]; quest'ultima è la metrica utilizzata nei capitoli successivi per valutare la qualità delle immagini generate.

Nel Capitolo 3 vengono descritti il funzionamento e la struttura dei VAE, sia nel training che in fase di generazione.

Nel Capitolo 4 viene mostrato il modello del Two-Stage VAE, l'estensione a due stadi discussa in [5]; vengono inoltre mostrati i dati generati utilizzando questa architettura, insieme ai valori della FID calcolati per le immagini prodotte.

Nel Capitolo 5 viene proposta l'estensione condizionale del modello a due stadi; anche qui vengono mostrati alcuni esempi di generazione di dati con questa architettura.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco delle implementazioni

Capitolo 1

Fondamenti di deep learning

1.1 Reti neurali

Una rete neurale è un sofisticato mezzo di approssimazione di funzioni, rappresentabile con una struttura a grafo orientato; i nodi, o neuroni, sono le unità basilari della rete, composte da una serie di input pesati a cui viene applicata successivamente una funzione di attivazione non lineare; i neuroni sono quindi fondamentalmente unità computazionali che ricevono una serie di input, applicano una serie di operazioni lineari su di essi, e restituiscono un output a cui viene applicata un'operazione non lineare.

La struttura della rete neurale è stratificata, dove ogni strato, o layer, è composto da una serie di neuroni che ricevono in input gli output del layer precedente, e forniscono l'input per il successivo; si ha quindi che una rete neurale minimale è composta da tre layer, chiamati *input layer*, *hidden layer* e *output layer*, come illustrato in Figura 1.1; i layer qui illustrati sono completamente connessi (o densi), i.e. ogni loro nodo è collegato a tutti i nodi del layer successivo.
Il grafo rappresentativo della rete neurale è aciclico nel caso di reti feed-forward, e contiene cicli nel caso di reti neurali ricorrenti.

Il risultato della calibrazione, o training, di una rete neurale, è quindi un'approssimazione di una qualche funzione, con un numero generalmente molto alto di parametri.

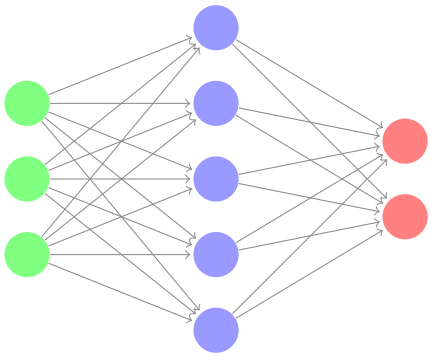

Figura 1.1: Esempio di struttura di una rete neurale. Da sinistra a destra: input layer, hidden layer e output layer.

1.2 Reti neurali convoluzionali

Un tipo particolare di rete neurale è rappresentato dalle reti neurali convoluzionali (CNN), usate principalmente per elaborazione di immagini, ma applicabili anche a tipologie di input differenti (e.g. audio).
La particolarità di questo genere di rete è la presenza di layer convoluzionali, nei quali non tutti i nodi sono connessi con lo strato successivo (diversamente dai layer densi).

Supponendo di avere un'immagine come dato di input, l'operazione di convoluzione eseguita da questi layer consiste nell'applicare un filtro (o *kernel*, una matrice di pesi, generalmente 3x3) ad un'area dell'immagine, ovvero nell'effettuare una moltiplicazione elemento per elemento tra le due matrici (il filtro e i valori dei pixel), e sommare i prodotti tra di loro in un valore di output.
Questa operazione viene ripetuta per tutte le aree dell'immagine, producendo una matrice di valori in output (Figura 1.2).

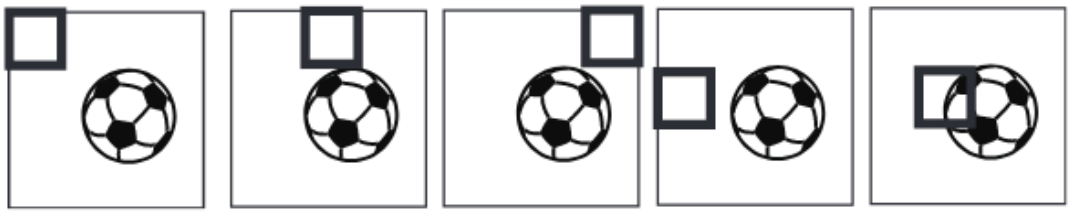

Figura 1.2: Spostamento della matrice lungo tutta l'immagine (da CS229, Stanford).

Nei layer convoluzionali si usa in realtà un insieme di filtri, i quali trasformano le immagini di input in immagini di output; se prendiamo un layer con 4 filtri, vediamo come ognuno di essi produce un'immagine in output; partendo da un'immagine 28x28, produciamo quindi (con adeguato padding) 4 immagini 28x28, con dimensione totale dell'output 28x28x4.
Come risultato, abbiamo un totale di $3 \times 3 \times 4 = 36$ pesi, ricordando che i filtri hanno dimensione 3x3.

Questo approccio presenta in molti casi due grandi vantaggi rispetto ai layer densi: prima di tutto, il numero di operazioni da svolgere è decisamente inferiore, importante nel caso appunto delle immagini: un'immagine 28x28, con un solo canale di colore, consiste in 784 valori di input; con un layer intermedio a 1024 nodi ci sarebbero più di 8×10^5 pesi da calibrare, decisamente superiori rispetto ai 36 del layer precedentemente illustrato; l'altro vantaggio è la possibilità di considerare i pixel in relazione a quelli circostanti, permettendo di valutare aree di spazio piuttosto che singoli punti, che nelle immagini risulta fondamentale.

Oltre ai layer convoluzionali è da notare che in questo genere di modelli si trovano anche layer di *pooling*, il cui scopo è di ridurre le dimensioni dell'input passato attraverso la rete; in breve, con un layer di pooling di dimensione n, si attraversa l'immagine in blocchi nxn, selezionando in ogni blocco solo un determinato valore, a seconda di una qualche funzione che determina il tipo di pooling (e.g. con maxpooling si estrae il valore massimo, con minpooling si estrae il valore minimo, e così via).
Il pooling divide altezza e larghezza dell'immagine per le dimensioni del blocco.

1.3 Apprendimento

Quando si parla di apprendimento, allenamento o training si intende la calibrazione dei parametri della rete neurale sulla base dei dataset di riferimento, e di conseguenza dei dati forniti in input al modello; l'algoritmo alla base della calibrazione è la cosiddetta retropropagazione dell'errore (o *backpropagation*).

La fase iniziale dell'apprendimento consiste nel passaggio dei dati attraverso la rete, ovvero, nell'applicazione layer per layer delle trasformazioni associate ai vari neuroni, prendendo come input i risultati del layer precedente, e fornendo in output gli input per il layer successivo; i parametri iniziali della rete sono generalmente casuali.
Successivamente al passaggio dei dati attraverso la rete, questa produce un valore di output confrontabile con un qualche risultato atteso; questo confronto viene effettuato secondo i termini di un'adeguata *funzione di loss*, che fornisce una misura dell'errore commesso dal modello; una volta valutato, questo viene per l'appunto propagato all'indietro nella rete, distribuendolo tra i neuroni in base al loro contributo verso l'errore totale.
I pesi della rete vengono quindi aggiornati in modo da minimizzare la loss, tramite la tecnica della discesa del gradiente (*gradient descent*): si calcola la derivata prima (o gradiente) della funzione di loss, e in base al suo andamento si modificano in piccoli incrementi i pesi, in modo da raggiungere un punto di minimo globale della funzione, nel tentativo di renderla nulla.

Il processo di training della rete è scandito in *epoch*; un epoch viene terminato quando tutti i dati di input vengono passati attraverso la rete (e di conseguenza viene ridistribuita "all'indietro" la loss) una volta; a causa delle dimensioni elevate dei dataset normalmente utilizzati, l'input viene generalmente diviso in *batch* di piccole dimensioni rispetto alla totalità dei dati, portando l'epoch ad essere una serie di iterazioni sulla rete, fino ad esaurire il dataset, per poi ricominciare all'epoch successivo; nel caso quindi di un dataset con 50.000 immagini è possibile ad esempio dividere l'input in batch da 100 elementi, avendo un epoch composto da 500 iterazioni sulla rete.

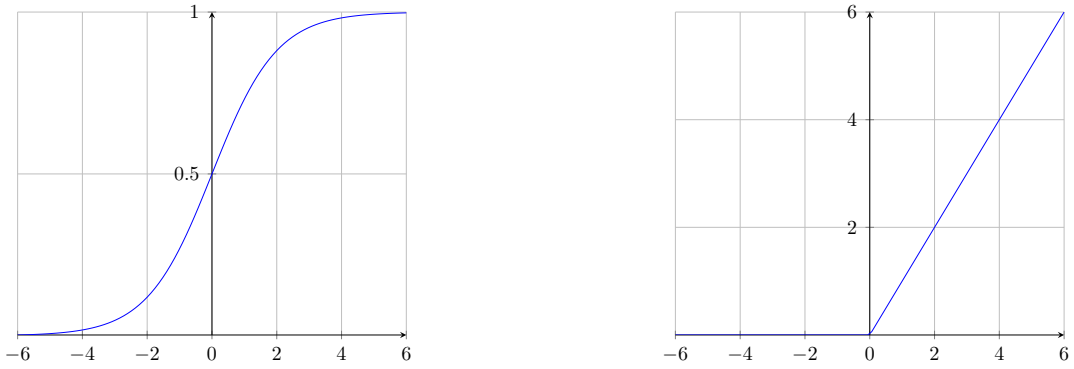

Figura 1.3: Grafici delle funzioni di attivazione; a sinistra sigmoide, a destra ReLU.

Si ricorda infine che le funzioni di attivazione dei neuroni, che determinano l'output (e quindi l'input per il layer successivo), devono essere non lineari; questo perché altrimenti si avrebbe un'unica combinazione lineare degli input come output della rete, riducendosi in pratica ad un unico layer; per esprimere (ed approssimare) relazioni non lineari tra i dati occorrono quindi funzioni non lineari tra un layer e l'altro.
Alcuni esempi di funzioni di attivazione sono la sigmoide $\frac{1}{1+e^{-x}}$ e la ReLU (***Re**ctified **L**inear **U**nit*), ovvero $max(x, 0)$; la prima produce output tra 0 e 1, la seconda produce output ≥ 0, come mostrato in Figura 1.3.

Capitolo 2

Misura delle prestazioni di metodi generativi

Sebbene gli approcci finora descritti siano spesso utilizzati per compiti di classificazione (e.g. riconoscimento di oggetti), esiste la possibilità di sfruttare queste tecniche a scopi generativi, ovvero per produrre nuovi dati come immagini, audio o contenuti testuali; nel contesto di questa tesi viene considerata solo la generazione di immagini.

Valutare le prestazioni di un modello per classificare dati consiste spesso nel calcolo dell'accuratezza del suddetto modello, generalmente considerando il numero di errori commessi nella classificazione dei dati di test; valutare la qualità delle immagini generate d'altro canto è spesso più difficile, e non esiste ancora una metrica universalmente accettata; di seguito ne sono presentate due, Inception Score (IS)[18] e Fréchet Inception Distance (FID)[10], quest'ultima usata nei Capitoli 4 e 5 per dare una misura dei risultati ottenuti.

2.1 Inception Score

Per calcolare l'Inception Score si applica una rete Inception-v3[20] (una CNN utilizzata per compiti di classificazione di immagini), allenata precedentemente su database ImageNet[6], alle immagini generate, confrontando poi la distribuzione condizionata e quella marginale per le categorie inferite dalla rete; una distribuzione condizionata molto prevedibile risulta in immagini realistiche (a partire dall'immagine, è facile determinarne la categoria), mentre una distribuzione marginale poco prevedibile, o uniforme, risulta in immagini diverse tra loro.

Detta D_{KL} la divergenza Kullback-Leibler tra due distribuzioni P e Q, definita come:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{2.1}$$

Si ha:

$$\mathrm{IS} = \exp\Big(\mathbb{E}_{x\sim p_g} D_{\mathrm{KL}}(p(y|x)\|p(y))\Big) \tag{2.2}$$

Un IS più alto rappresenta una maggiore D_{KL} tra le due distribuzioni, e quindi immagini generate di migliore qualità.

2.2 Fréchet Inception Distance

Il principale difetto della metrica IS è la rappresentazione delle prestazioni in caso di una sola immagine generata; in questo caso infatti $p(y)$ risulta uniforme, nonostante la diversità sia molto bassa.
Un miglioramento proposto è l'approccio della Fréchet Inception Distance, che utilizza sempre Inception-v3, in questo caso però per estrarre feature da un suo layer intermedio, mettendo a confronto quelle derivanti da immagini reali e quelle derivanti da immagini generate; non si ha quindi una valutazione isolata dei dati prodotti, come avveniva nel calcolo dell'Inception Score, bensì una comparazione tra questi e dati reali. Questa metrica è basata sulla distanza di Fréchet, una misura della similarità tra due curve.

Si considerano quindi i valori delle attivazioni di un layer intermedio di Inception-v3, per poi calcolare la distanza di Fréchet tra le due gaussiane multivariate che modellano le distribuzioni dei suddetti valori; il layer generalmente scelto è l'ultimo layer di pooling della rete.
Ponendo $X_r \sim \mathcal{N}(\mu_r, \Sigma_r)$ e $X_g \sim \mathcal{N}(\mu_g, \Sigma_g)$ le attivazioni del layer, rispettivamente per immagini reali e generate, si ha:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{\frac{1}{2}}) \tag{2.3}$$

Una FID minore rappresenta migliori immagini generate, ovvero immagini più simili ad esempi reali; non incontra inoltre lo stesso problema di IS, poiché un set composto da una sola immagine ripetuta avrà una distribuzione molto diversa rispetto ad uno composto da immagini reali.

Capitolo 3

Variational Autoencoder

Il metodo generativo di interesse in questa tesi è quello del Variational Autoencoder, o VAE ([13], [14]); prima di illustrarne il funzionamento, vengono di seguito considerate brevemente differenze e similarità tra questo modello ed un generico autoencoder.

3.1 Autoencoder e VAE

Un autoencoder è una rete composta da due sottostrutture, un encoder e un decoder; l'encoder riceve in input i dati e ne produce una rappresentazione compressa, il decoder riceve in input la suddetta rappresentazione e ricostruisce i dati di origine; l'obiettivo fondamentale di un autoencoder è quindi quello di apprendere una codifica efficiente di un insieme di dati, e generalmente di produrne una ricostruzione che rispetti alcune proprietà fondamentali; ad esempio, nel caso dei Denoising Autoencoder (DAE), la ricostruzione consiste nell'estrazione del dato originale da un input parzialmente corrotto (non copiando quindi interamente il dato fornito, ma estraendone caratteristiche utili).

A differenza di un autoencoder classico, un VAE è invece un modello generativo, utilizzato per la produzione di nuovi dati (ad esempio immagini); la principale carat-

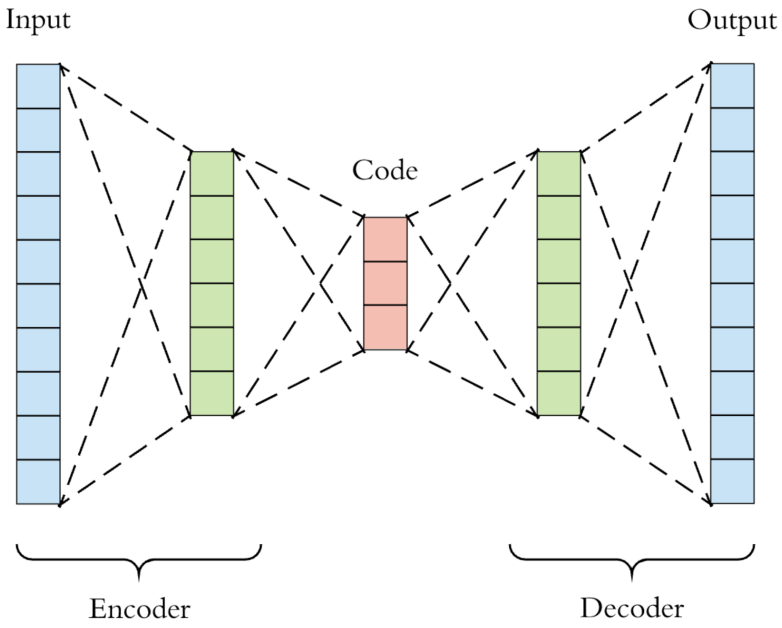

Figura 3.1: Struttura di base di un autoencoder (da Comprehensive Introduction to Autoencoders); "code" indica la rappresentazione compressa dell'informazione.

teristica che lo distingue da un normale autoencoder è la forma della rappresentazione compressa (o spazio latente): l'encoder di un VAE genera una media (μ) e una varianza (Σ) per ogni variabile della rappresentazione, per poi fornire in input al decoder i valori risultanti dal campionamento della Gaussiana di media μ e varianza Σ (si veda anche [7]).

3.2 Funzionamento

In fase di generazione si campiona lo spazio latente, per poi dare i valori ottenuti come input al decoder; di conseguenza, per ottenere un buon modello generativo occorre poter campionare efficacemente lo spazio latente. Si sceglie quindi un *prior*, una distribuzione più semplice a cui si cerca di far convergere la rappresentazione dei dati, e da cui campionare i valori da fornire al decoder.
Da ciò deriva che più la forma dello spazio latente si avvicinerà a quella del prior, più

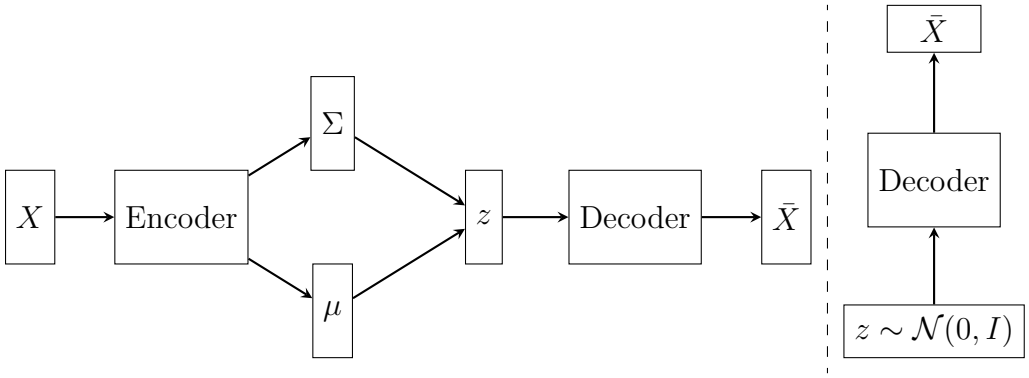

Figura 3.2: A sinistra, struttura di un VAE: l'encoder produce una media μ e una varianza Σ, tramite le quali viene campionato z, l'input del decoder; a destra, fase di generazione in un VAE: z viene campionato da una Gaussiana $\mathcal{N}(0, I)$, e viene poi usato come input del decoder per produrre un output $\bar{X}$.

probabile sarà campionare punti effettivamente significativi per il decoder in fase di generazione; al contrario, se lo spazio latente avrà una forma molto distante da quella del prior, sarà molto improbabile che i punti campionati da quest'ultimo corrispondano a valori significativi nella rappresentazione dei dati. Per esprimere la necessità di far convergere la forma dello spazio latente verso quella del prior si può utilizzare la divergenza Kullback-Leibler tra la distribuzione del primo e quella del secondo, ottenendo un termine da minimizzare attraverso la calibrazione della rete.

Abbiamo quindi che in fase di training della rete neurale dovremo ottimizzare sia la ricostruzione delle immagini in input (in modo da apprendere una rappresentazione di immagini "realistiche", ovvero simili a quelle del dataset), sia la distanza tra il prior e la forma dello spazio latente; si utilizza allora la somma di due funzioni di loss separate: la D_{KL} tra il prior (generalmente una Gaussiana $\mathcal{N}(0, I)$) e la distribuzione delle variabili latenti, e l'errore quadratico medio (MSE, *mean squared error*) tra i dati di input e quelli ricostruiti; la loss totale assume quindi la seguente forma:

$$\mathcal{L} = D_{\mathrm{KL}}(Q(z|X) \| P(z)) + mse(x, \bar{x}) \tag{3.1}$$

Con $P(z)$ prior, $Q(z|X)$ distribuzione delle variabili latenti z avendo i dati X, e $mse(x, \bar{x})$ scarto quadratico medio tra i dati reali e quelli ricostruiti.

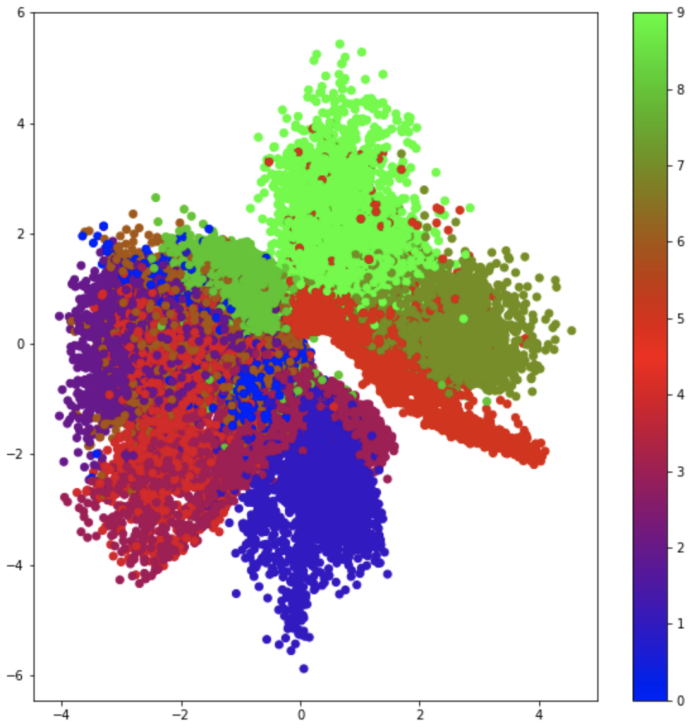

Figura 3.3: Rappresentazione di uno spazio latente a due dimensioni; ogni colore indica una diversa classe nel dataset utilizzato, in questo caso fashion-MNIST[22] (da Comprehensive Introduction to Autoencoders).

La formula effettiva per il calcolo della loss può comprendere anche un termine di ridimensionamento per una delle due funzioni ([4], [11]), generalmente con l'intento di calibrare l'impatto della D_{KL} sul processo di ottimizzazione; anche nel modello descritto nel Capitolo 4 è presente un approccio simile, con un fattore γ allenabile presente all'interno della funzione di loss.
Definiendo $H(P, Q)$ la cross-entropy tra due distribuzioni di probabilità come:

$$H(P, Q) = \mathbb{E}_{x\sim P}[-\log Q(x)] \tag{3.2}$$

possiamo esprimere la D_{KL} come:

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) &= \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \\ &= \mathbb{E}_{x\sim P}[-\log Q(x)] - \mathbb{E}_{x\sim P}[-\log P(x)] \\ &= H(P, Q) - H(P) \end{aligned} \tag{3.3}$$

Come mostrato in [2] si può osservare che, considerando la media di $D_{\mathrm{KL}}(Q(z|X)\|P(z))$ su tutti i dati di input, vale la seguente uguaglianza:

$$\mathbb{E}_X\, D_{\mathrm{KL}}(Q(z|X)\|P(z)) = H(Q(z), P(z)) - \mathbb{E}_X\, H(Q(z|X)) \tag{3.4}$$

Minimizzare $D_{\mathrm{KL}}(Q(z|X)\|P(z))$ porta quindi a ridurre anche $H(Q(z), P(z))$; poiché questa risulta minima quando le due distribuzioni coincidono, il termine della D_{KL} nella loss spinge allora l'intero spazio latente (distribuito secondo $Q(z)$) ad assumere la forma descritta dal prior.

Il risultato del processo di ottimizzazione sarà, idealmente, uno spazio in cui ogni variabile latente codificherà una qualche caratteristica o informazione utile delle immagini di input, e quindi nel quale per ogni punto del dataset esisterà una configurazione delle variabili latenti che possa venire decodificata per ricostruire un'immagine simile a quella data; campionando questo spazio e decodificando i valori ottenuti si potranno quindi generare immagini nuove, ricostruite a partire da una combinazione di caratteristiche di immagini già esistenti.

3.3 Variabili inattive

Un fenomeno interessante che si può osservare nel training dei VAE è quello del collasso delle variabili latenti([3], [4]), ovvero, la presenza di variabili latenti che vengono ignorate in fase di generazione in quanto non codificano alcuna informazione utile per il modello; le variabili di questo tipo vengono dette inattive, e sono caratterizzate da una varianza con valore medio molto vicino a 1.
Questo fenomeno tende a verificarsi con l'aumento della dimensione dello spazio latente: infatti, un maggior numero di variabili latenti non viene necessariamente sfruttato dalla rete per rappresentare i dati di input, portando quindi un quantitativo sempre maggiore di queste variabili a venire ignorato in fase generativa; è anche interessante osservare come un altro fattore che favorisce questo fenomeno sia la presenza della D_{KL} all'interno della funzione di loss: infatti, una variabile con media 0 e varianza 1 porterà al minimo

il valore della D_{KL}, pur sacrificando nei fatti l'utilità della variabile dal punto di vista della codifica; una Kullback-Leibler con peso maggiore all'interno della loss avrà quindi generalmente come conseguenza secondaria l'aumento delle variabili inattive.

Capitolo 4

Two-Stage Variational Autoencoder

Nonostante i vantaggi presentati dai VAE dal punto di vista della stabilità del training e della semplicità della generazione di contenuti, i risultati prodotti da questo approccio tendono ad essere poco soddisfacenti in termini di qualità delle immagini; in questa sezione e nella successiva viene analizzato un modello derivante dai VAE, il Two-Stage Variational Autoencoder ([5]) studiato per migliorarne le capacità generative.

4.1 Funzionamento

L'idea di base del Two-Stage VAE è che, anche nel caso in cui un singolo autoencoder dovesse riuscire a rappresentare efficacemente la distribuzione dei dati reali, il suo codice latente potrebbe non rispettare a pieno le proprietà desiderate (nello specifico, la vicinanza alla distribuzione normale); per ovviare a questa limitazione si utilizza un secondo VAE, avente come dati in input le variabili latenti del primo.

Il processo di generazione avviene quindi sempre campionando una Gaussiana $\mathcal{N}(0, 1)$, e fornendo questa volta il risultato come input per il secondo decoder; il risultato sarà utilizzato poi come input per il primo decoder.
Il compito del secondo stage è quindi di creare uno spazio latente ancora più vicino al

prior ed ottimizzare la generazione a partire dal campionamento, in quanto l'output del secondo decoder sarà generalmente un insieme di valori latenti significativi per il primo VAE.

4.2 Implementazione

Nelle seguenti sezioni verrà descritta la struttura del Two-Stage VAE dal punto di vista architetturale, seguita da un'implementazione in Python[21], nello specifico utilizzando la libreria Tensorflow[1]; in questo capitolo sarà mostrata la struttura originale del modello, di cui si fa riferimento in [5].

4.2.1 Primo stage

La struttura del primo stage è basata sull'architettura di una ResNet[9] (*residual neural network*); come mostrato in Figura 4.1, il blocco fondamentale che compone la rete (*ScaleBlock*) è infatti a sua volta formato da due *ResBlock*, ovvero blocchi residui; questi sono caratterizzati dalla presenza di uno "shortcut", ovvero, una connessione diretta tra l'input e l'output, che ignora i layer interni al blocco; in questo modo il risultato della computazione diventa composto sia dall'input originale, sia dall'output dell'ultimo layer interno. L'idea di base di questo tipo di architettura è di evitare il problema della scomparsa del gradiente[12], che tende a ostacolare il training delle reti neurali una volta superata una certa profondità.

L'encoder del primo stage è quindi composto da un layer convoluzionale iniziale, seguito da coppie di ScaleBlock e layer di downsampling, ovvero layer convoluzionali che dimezzano le dimensioni dell'input in altezza e in larghezza (un'immagine di input con risoluzione 32x32 diventerà quindi un output con risoluzione 16x16); dopo N ScaleBlock (3 in caso si stesse lavorando su dataset cifar10[15], 4 in caso si stesse lavorando su dataset Celeba[17]), l'output viene ridotto a un vettore, per poi venire passato in un altro ScaleBlock, nel quale i layer convoluzionali sono stati sostituiti da layer densi;

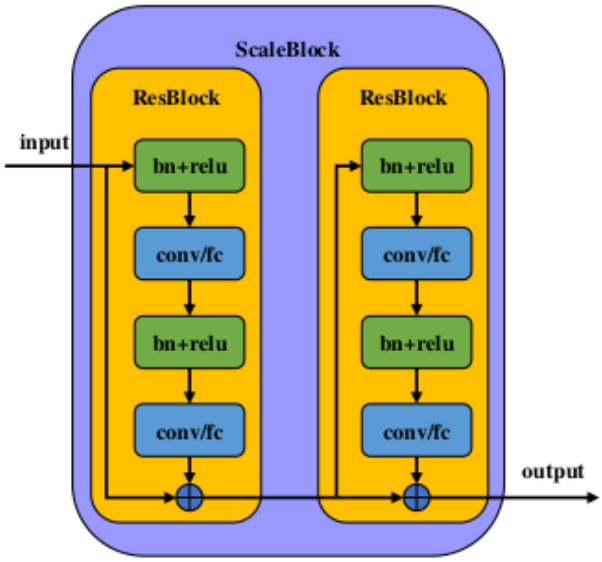

Figura 4.1: Struttura del blocco fondamentale del primo stage, lo *ScaleBlock*, come mostrato in [5].

l'output di questo blocco viene poi utilizzato per produrre la media e la varianza per le variabili latenti (μ_x e σ_x, dove x fa riferimento alle immagini); il codice latente ha dimensione 64 in caso si utilizzi dataset cifar10, e 128 nel caso si utilizzi Celeba.

Il decoder ha una struttura speculare a quella dell'encoder: l'input (il codice latente) viene fatto passare per un layer denso, e successivamente rimodellato in un'immagine di risoluzione 2x2; questa viene poi fatta passare per una serie di layer di upsampling (ovvero layer convoluzionali che raddoppiano le dimensioni dell'immagine in larghezza ed altezza) seguiti da ScaleBlock; il risultato di questa serie di blocchi viene poi usato come input per un layer convoluzionale, che converte l'immagine al numero di canali desiderati (1 per immagini in bianco e nero, 3 per immagini a colori); questa diventa poi l'output del decoder, denominato $\hat{x}$.

L'architettura usata per i risultati ottenuti in questa tesi è lievemente diversa da quella descritta in [5] e nell'immagine; per ragioni di brevità del training e di coerenza

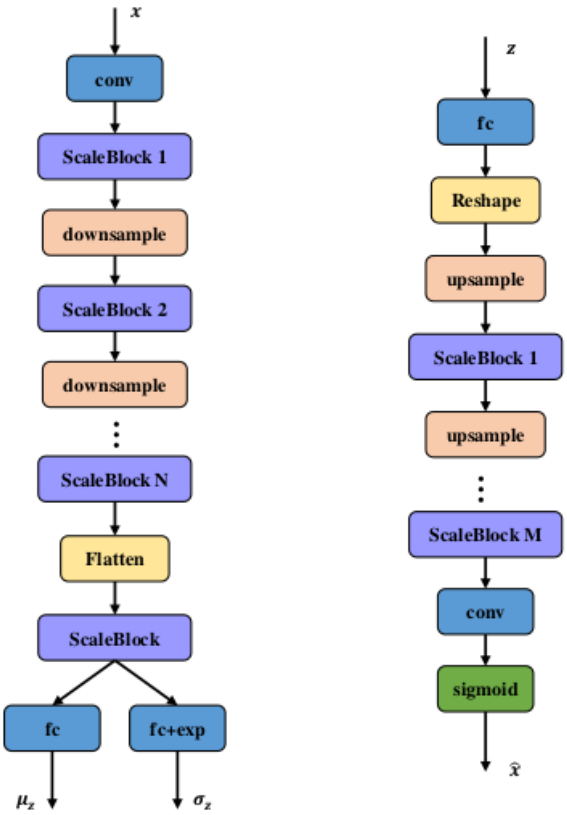

Figura 4.2: Architettura del primo stage, come mostrato in [5].

con il codice originale sono stati utilizzati blocchi contenenti un solo *ResBlock* ciascuno, invece di due; i risultati osservati sono comunque comparabili con quelli ricavati dall'architettura originale.
L'implementazione del primo stage è mostrata nelle Implementazioni 4.1 e 4.2.

4.2.2 Secondo stage

Il secondo stage ha una struttura comparativamente molto semplice rispetto al primo: sia l'encoder che il decoder sono composti da tre layer densi; dal punto di vista dimensionale, sia l'output dell'encoder che quello del decoder corrispondono alle dimensioni dello spazio latente del primo stage.

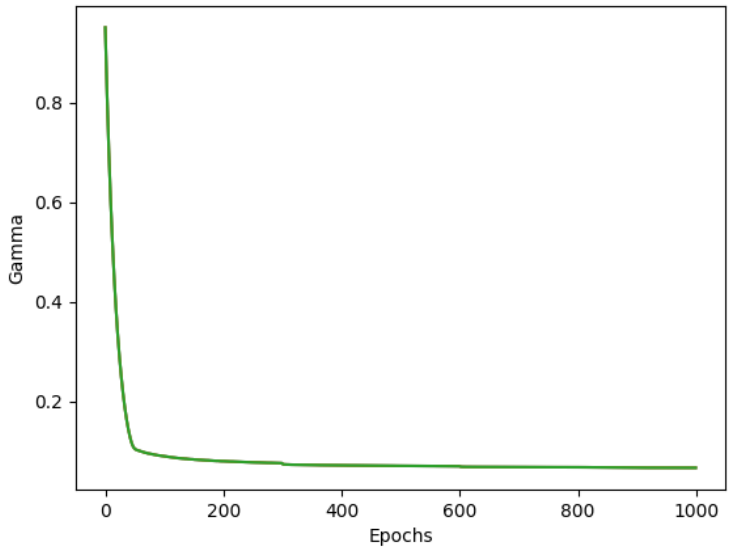

Figura 4.3: Andamento di γ durante il training del primo stage su dataset cifar10.

L'implementazione del secondo stage è mostrata nell'Implementazione 4.3.

4.2.3 Loss

La funzione di loss utilizzata dal modello presenta una differenza principale con quella descritta nel Capitolo 3, ovvero la presenza di un parametro γ calibrabile; l'utilità di questo parametro è fondamentalmente quella di dare alla ricostruzione un peso maggiore rispetto alla D_{KL}, adattando però questo peso durante il training; approcci simili relativi alla calibrazione del peso della Kullback-Leibler sono presenti in [4] e [11]. La formula esatta per il calcolo della loss e per la sua ottimizzazione è presente in [5].
Durante il training è possibile osservare che il modello tende a minimizzare γ, come mostrato in Figura 4.3; di conseguenza, la D_{KL} ha un impatto maggiore sulla loss all'inizio del training, ma il suo peso cala rapidamente dopo i primi epoch, portando la ricostruzione ad avere maggiore rilevanza nell'obiettivo da ottimizzare.
Il codice per il calcolo della loss è mostrato nell'Implementazione 4.4.

FID	cifar10	Celeba
Ricostruzione	52.94131	49.10653
Generazione (primo stage)	75.93841	68.03379
Generazione (secondo stage)	71.89105	58.44788

Tabella 4.1: FID calcolata sui risultati del Two-Stage VAE con dataset cifar10 e Celeba; i risultati migliorano utilizzando anche il secondo stage, come previsto.

4.3 Risultati sperimentali

L'utilizzo del Two-Stage VAE porta a buoni risultati sia dal punto di vista ricostruttivo, sia soprattutto dal punto di vista generativo; si può notare inoltre, soprattutto su Celeba, l'effetto positivo che ha sulla generazione l'utilizzo del secondo stage, non solo visivamente ma anche a livello di FID (Tabella 4.1). In Figura 4.4 sono mostrati i risultati ricostruttivi sia su dataset cifar10 che su Celeba; in Figura 4.5 e Figura 4.6 sono mostrati invece i risultati generativi rispettivamente sul primo e sul secondo dataset.
Su dataset cifar10 il training ha avuto una durata di 1000 epoch per il primo e di 2000 epoch per il secondo stage; su dataset Celeba la durata è stata rispettivamente di 120 e 300 epoch per il primo e il secondo stage.

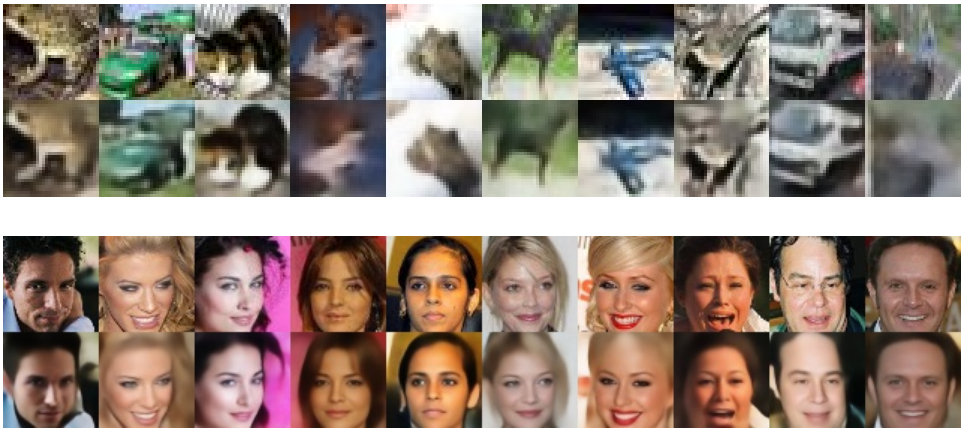

Figura 4.4: Risultati ricostruttivi su cifar10 (in alto) e Celeba (in basso).

Figura 4.5: Risultati generativi su cifar10; a sinistra i risultati del primo stage, a destra quelli del secondo stage.

Figura 4.6: Risultati generativi su Celeba; a sinistra i risultati del primo stage, a destra quelli del secondo stage.

```
def build_encoder1(self):
    with tf.variable_scope('encoder'):
        dim = self.base_dim
        y = tf.layers.conv2d(self.x, dim, self.kernel_size, 1, 'same', name='↘
            conv0')
        for i in range(self.num_scale):
            y = scale_block(y, dim, self.is_training, 'scale'+str(i), self.↘
                block_per_scale, self.depth_per_block, self.kernel_size)
            if i != self.num_scale - 1:
                dim *= 2
                y = downsample(y, dim, self.kernel_size, 'downsample'+str(i))
        y = tf.reduce_mean(y, [1, 2])
        y = scale_fc_block(y, self.fc_dim, 'fc', 1, self.depth_per_block)

        self.mu_z = tf.layers.dense(y, self.latent_dim)
        self.logsd_z = tf.layers.dense(y, self.latent_dim)
        self.sd_z = tf.exp(self.logsd_z)
        self.z = self.mu_z + tf.random_normal([self.batch_size, self.latent_dim])↘
            * self.sd_z
```

Implementazione 4.1: Encoder del primo stage; `num_scale` è pari a 3 o 4, a seconda del dataset utilizzato, `fc_dim` vale 512 (dimensione dell'output dei layer densi usati al punto di quelli convoluzionali nell'ultimo ScaleBlock), e `block_per_scale` vale 1, come descritto precedentemente nel Capitolo 4.

```
def build_decoder1(self):
    desired_scale = self.x.get_shape().as_list()[1]
    scales, dims = [], []
    current_scale, current_dim = 2, self.base_dim
    while current_scale <= desired_scale:
        scales.append(current_scale)
        dims.append(current_dim)
        current_scale *= 2
        current_dim = min(current_dim*2, 1024)
    assert(scales[-1] == desired_scale)
    dims = list(reversed(dims))
    with tf.variable_scope('decoder'):
        y = self.z
        data_depth = self.x.get_shape().as_list()[-1]
        fc_dim = 2 * 2 * dims[0]
        y = tf.layers.dense(y, fc_dim, name='fc0')
        y = tf.reshape(y, [-1, 2, 2, dims[0]])
        for i in range(len(scales)-1):
            y = upsample(y, dims[i+1], self.kernel_size, 'up'+str(i))
            y = scale_block(y, dims[i+1], self.is_training, 'scale'+str(i), self.↘
                block_per_scale, self.depth_per_block, self.kernel_size)
        y = tf.layers.conv2d(y, data_depth, self.kernel_size, 1, 'same')

        self.x_hat = tf.nn.sigmoid(y)
        self.loggamma_x = tf.get_variable('loggamma_x', [], tf.float32, tf.↘
            zeros_initializer())
        self.gamma_x = tf.exp(self.loggamma_x)
```

Implementazione 4.2: Decoder del primo stage; le dimensioni degli output dei vari layer partono da una risoluzione 2x2, per poi raddoppiare altezza e larghezza ad ogni passaggio di upsampling, fino alla dimensione desiderata (32x32 nel caso di cifar10, 64x64 nel caso di Celeba).

```
def build_encoder2(self):
    with tf.variable_scope('encoder'):
        t = self.z
        for i in range(self.second_depth):
            t = tf.layers.dense(t, self.second_dim, tf.nn.relu, name='fc'+str(i))
        t = tf.concat([self.z, t], -1)

        self.mu_u = tf.layers.dense(t, self.latent_dim, name='mu_u')
        self.logsd_u = tf.layers.dense(t, self.latent_dim, name='logsd_u')
        self.sd_u = tf.exp(self.logsd_u)
        self.u = self.mu_u + self.sd_u * tf.random_normal([self.batch_size, self.↘
            latent_dim])

def build_decoder2(self):
    with tf.variable_scope('decoder'):
        t = self.u
        for i in range(self.second_depth):
            t = tf.layers.dense(t, self.second_dim, tf.nn.relu, name='fc'+str(i))
        t = tf.concat([self.u, t], -1)

        self.z_hat = tf.layers.dense(t, self.latent_dim, name='z_hat')
        self.loggamma_z = tf.get_variable('loggamma_z', [], tf.float32, tf.↘
            zeros_initializer())
        self.gamma_z = tf.exp(self.loggamma_z)
```

Implementazione 4.3: Encoder e decoder del secondo stage; `self.u` indica il codice latente, derivato da `self.z`, ovvero il codice latente del primo stage, e input del secondo.

```
HALF_LOG_TWO_PI = 0.91893

self.kl_loss1 = tf.reduce_sum(tf.square(self.mu_z) + tf.square(self.sd_z) - 2 * ↘
    self.logsd_z - 1) / 2.0 / float(self.batch_size)
self.gen_loss1 = tf.reduce_sum(tf.square((self.x - self.x_hat) / self.gamma_x) / ↘
    2.0 + self.loggamma_x + HALF_LOG_TWO_PI) / float(self.batch_size)
self.loss1 = self.kl_loss1 + self.gen_loss1

self.kl_loss2 = tf.reduce_sum(tf.square(self.mu_u) + tf.square(self.sd_u) - 2 * ↘
    self.logsd_u - 1) / 2.0 / float(self.batch_size)
self.gen_loss2 = tf.reduce_sum(tf.square((self.z - self.z_hat) / self.gamma_z) / ↘
    2.0 + self.loggamma_z + HALF_LOG_TWO_PI) / float(self.batch_size)
self.loss2 = self.kl_loss2 + self.gen_loss2
```

Implementazione 4.4: Loss calcolata per il training del modello; le variabili indicate con `mu_`, `sd_` e `logsd_` indicano rispettivamente la media, la varianza e il logaritmo della varianza del codice latente (`z` nel caso del primo stage e `u` nel caso del secondo stage).

Capitolo 5

Conditional Two-Stage Variational Autoencoder

5.1 Conditional Variational Autoencoder

Una limitazione nel processo generativo dei VAE è l'impossibilità di scegliere i dati da generare; nello specifico, non è possibile indicare una categoria definita di immagine da produrre, limitandosi quindi a campionare lo spazio latente e a decodificare i valori ottenuti.
L'approccio del CVAE[19], o Conditional Variational Autoencoder, mira a risolvere questo tipo di problematica, permettendo inoltre di ottimizzare la decodifica del codice latente grazie alla presenza di informazioni preesistenti riguardo al dato da interpretare.

La differenza principale tra questo modello e un normale VAE è quindi la presenza di una condizione, ovvero un dato categorico relativo all'immagine da codificare e decodificare; nel caso ad esempio di dataset MNIST[16], come mostrato di seguito, questa indica la cifra da generare.

Come mostrato in Figura 5.1, in fase di training la condizione (indicata con c)

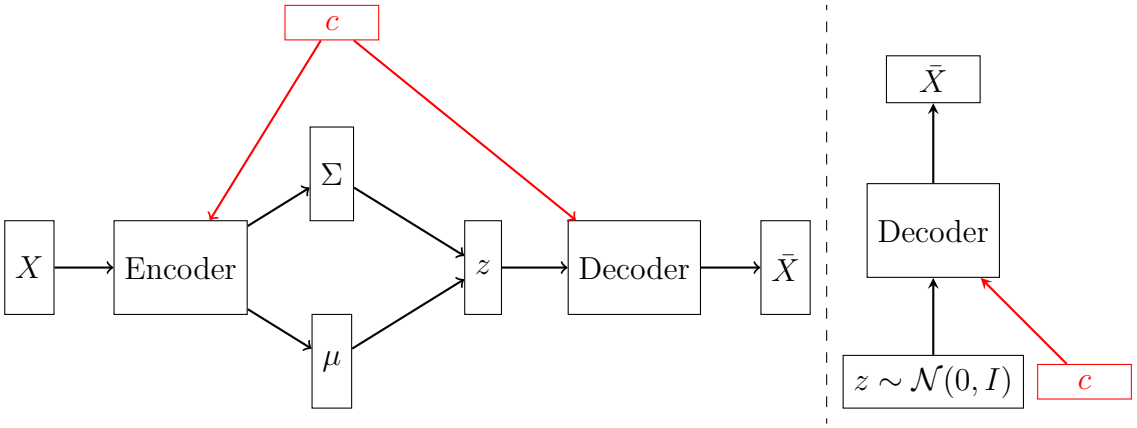

Figura 5.1: Struttura di un Conditional VAE; il modello è identico a quello mostrato in Figura 3.2, salvo per la condizione aggiunta in input sia all'encoder che al decoder (qui evidenziata in rosso); allo stesso modo, in fase di generazione viene concatenata la condizione all'input del decoder.

viene concatenata all'input sia dell'encoder, sia del decoder; di conseguenza la calibrazione della rete non dipende più solo dai dati, ma anche dall'informazione aggiuntiva fornita da c; indicando i dati con X, abbiamo quindi che (con riferimento al Capitolo 3) la distribuzione $Q(z|X)$ definita dall'encoder diventa $Q(z|X, c)$, evidenziando il condizionamento aggiunto dalla nuova informazione.
In fase di testing anche in questo caso viene campionato $z \sim \mathcal{N}(0, I)$, ma viene inoltre ad esso concatenato c, fornendo quindi un'informazione aggiuntiva al decoder anche in fase generativa.

5.2 Implementazione

Dal punto di vista dell'architettura del modello, la condizione viene aggiunta in posizioni diverse a seconda del componente che si sta considerando: nell'encoder del primo stage, la condizione viene aggiunta subito prima dell'ultimo ScaleBlock (quello caratterizzato da layer densi invece che convoluzionali), concatenata ai dati (a questo punto della rete in forma di vettore); nel decoder del secondo stage viene invece aggiunta

direttamente all'input, ovvero al codice latente derivato dall'encoder, prima che questo venga fatto passare per il primo layer denso; nell'encoder e nel decoder del secondo stage, la condizione viene nuovamente concatenata all'input, prima del primo layer, aggiungendo quindi direttamente alle variabili latenti l'informazione aggiuntiva della categoria.
Il resto del modello risulta invariato, sia per quanto riguarda i layer che per quanto riguarda la loss, la quale non viene modificata rispetto alla versione originale della rete; dal punto di vista funzionale, invece, cambiano le modalità di ricostruzione e generazione: in entrambe infatti viene fornita anche l'informazione condizionale, oltre ai normali dati in input, ovvero l'immagine da ricostruire nel primo caso, e il valore campionato da una normale standard nel secondo (come mostrato nei Capitolo 3 e Capitolo 4).
La condizione viene rappresentata come un vettore, il numero dei cui elementi varia in base al dataset considerato (un solo elemento nel caso di MNIST, e 40 elementi nel caso di Celeba); i valori degli elementi del vettore dipendono anch'essi dal dataset: per MNIST, il vettore ha un solo elemento con valore intero tra 0 e 9 (ad indicare la cifra che si desidera generare); per Celeba, i 40 elementi del vettore possono avere valore 1 o -1, a seconda che la caratteristica indicata, identificata dall'indice dell'elemento, sia vera o meno per l'immagine (le categorie di Celeba, ordinate per indice, sono mostrate in Tabella 5.1).

L'aggiunta della condizione al modello nel codice è mostrata nell'Implementazione 5.1.

5.3 Risultati sperimentali

Dal punto di vista della FID generativa l'estensione condizionale non sembra presentare miglioramenti degni di nota rispetto al modello originale (Tabella 5.2); su dataset MNIST è però stato implementato con successo il condizionamento dell'output specificando la cifra che si desidera generare. Su dataset Celeba questo stesso tentativo non ha invece avuto successo, probabilmente a causa della struttura più complessa delle categorie (10 categorie mutualmente esclusive per MNIST, 40 categorie combinabili per Celeba).

Figura 5.2: Generazione casuale (sinistra) e condizionata (destra) di immagini su dataset Celeba; l'estensione condizionale non è stata efficace nel produrre un miglioramento sui risultati.

In Figura 5.2 è riportata la generazione di immagini su Celeba, in cui la generazione condizionata produce risultati pressoché identici a quella casuale; in Figura 5.3 è invece riportata la generazione di cifre a piacere su MNIST confrontata con quella di immagini casuali. Tutti le immagini riportate in questo capitolo sono state generate utilizzando anche il secondo stage.
Su dataset Celeba il training ha avuto la stessa durata del modello descritto nel Capitolo 4 (120 e 300 epoch rispettivamente per primo e secondo stage); su dataset MNIST la durata è stata di 400 e 800 epoch rispettivamente per primo e secondo stage.

Indice	Etichetta
0	5_o_Clock_Shadow
1	Arched_Eyebrows
2	Attractive
3	Bags_Under_Eyes
4	Bald
5	Bangs
6	Big_Lips
7	Big_Nose
8	Black_Hair
9	Blond_Hair
10	Blurry
11	Brown_Hair
12	Bushy_Eyebrows
13	Chubby
14	Double_Chin
15	Eyeglasses
16	Goatee
17	Gray_Hair
18	Heavy_Makeup
19	High_Cheekbones
20	Male

Indice	Etichetta
21	Mouth_Slightly_Open
22	Mustache
23	Narrow_Eyes
24	No_Beard
25	Oval_Face
26	Pale_Skin
27	Pointy_Nose
28	Receding_Hairline
29	Rosy_Cheeks
30	Sideburns
31	Smiling
32	Straight_Hair
33	Wavy_Hair
34	Wearing_Earrings
35	Wearing_Hat
36	Wearing_Lipstick
37	Wearing_Necklace
38	Wearing_Necktie
39	Young

Tabella 5.1: Forma e significato delle categorie nel dataset Celeba: ogni etichetta può avere valore 1 (vero) o -1 (falso); questi valori vengono poi gestiti come un vettore, e vanno a determinare le caratteristiche complessive dell'immagine.

```
#encoder 1
y = tf.reduce_mean(y, [1, 2])
y = tf.concat([y, self.condition], axis = -1)
y = scale_fc_block(y, self.fc_dim, 'fc', 1, self.depth_per_block)
```

```
#decoder 1
with tf.variable_scope('decoder'):
    y = tf.concat([self.z, self.condition], axis = -1)
    data_depth = self.x.get_shape().as_list()[-1]
```

```
#encoder 2
def build_encoder2(self):
    with tf.variable_scope('encoder'):
        t = tf.concat([self.z, self.condition], axis = -1)
```

```
#decoder 2
def build_decoder2(self):
    with tf.variable_scope('decoder'):
        t = tf.concat([self.u, self.condition], axis = -1)
```

Implementazione 5.1: Implementazione della condizione nei vari componenti del modello; nell'ordine, encoder del primo stage, decoder del primo stage, encoder del secondo stage, decoder del secondo stage; sono mostrate le righe adiacenti per riferimento al codice del modello visto nel Capitolo 4.

FID generativa su Celeba	Two-Stage VAE	Estensione condizionale
Primo stage	68.03379	64.65185
Secondo stage	58.44788	57.113102

Tabella 5.2: Confronto della FID tra il modello originale e l'estensione condizionale; dal punto di vista generativo il modello non porta a miglioramenti degni di nota anche secondo la metrica della FID.

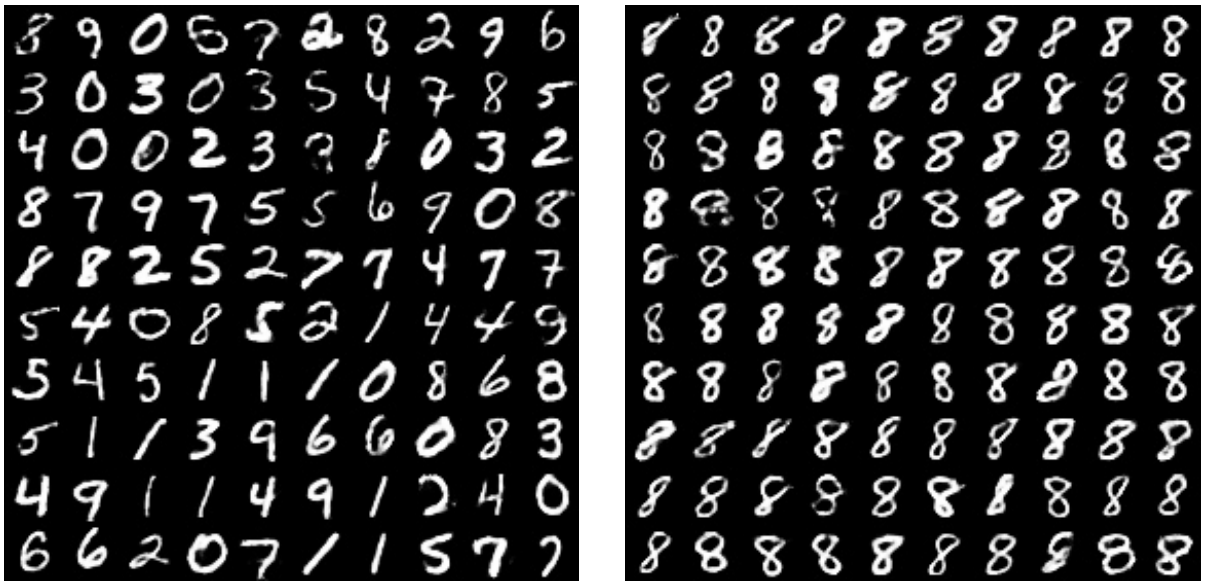

Figura 5.3: Generazione casuale (sinistra) e condizionata (destra) di cifre su dataset MNIST.

Conclusioni

In questa tesi è stata mostrata un'estensione a due stadi di un modello VAE, ideata per ottimizzare il processo generativo a partire dal codice latente; i risultati qui ottenuti sperimentalmente tendono a confermare quelli riportati in [5], anche utilizzando un'architettura più semplice rispetto a quella originalmente mostrata. È stato poi mostrato un esempio di estensione condizionale del modello precedentemente discusso, avente come obiettivo la generazione controllata di immagini, secondo categorie a piacere; questa nuova architettura ha mostrato ottimi risultati su dataset MNIST, ma si è rivelata inefficace su dataset Celeba, fornendo solo risultati lievemente migliori dal punto di vista generativo.

La generazione condizionata di dati a partire da Celeba è un possibile sviluppo di quanto esposto in questa tesi, in quanto combinata con la qualità generativa del modello originale potrebbe portare alla produzione di immagini di buona qualità con caratteristiche predeterminate, e potenzialmente anche a risultati migliori dal punto di vista della FID generativa.

Bibliografia

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] Andrea Asperti. «About Generative Aspects of Variational Autoencoders». In: *Proceedings of the Fifth International Conference on Machine Learning, Optimization, and Data Science – September 10-13, 2019 – Certosa di Pontignano, Siena – Tuscany, Italy*. LNCS (to appear). Springer, 2019.

[3] Andrea Asperti. «Sparsity in Variational Autoencoders». In: *Proceedings of the First International Conference on Advances in Signal Processing and Artificial Intelligence, ASPAI 2015, Barcelona, Spain, 20-22 March 2019*. 2019.

[4] Andrea Asperti. «Variational Autoencoders and the variable collapse phenomenon». In: *Sensors & Transducers, to appear* (2019).

[5] Bin Dai e David Wipf. «Diagnosing and Enhancing VAE Models». In: *arXiv e-prints*, arXiv:1903.05789 (2019).

[6] J. Deng et al. «ImageNet: A large-scale hierarchical image database». In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255.

[7] Carl Doersch. «Tutorial on Variational Autoencoders». In: *arXiv e-prints*, arXiv:1606.05908 (2016).

[8] Ian J. Goodfellow et al. «Generative Adversarial Networks». In: *arXiv e-prints*, arXiv:1406.2661 (2014).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren e Jian Sun. «Deep Residual Learning for Image Recognition». In: *arXiv e-prints*, arXiv:1512.03385 (2015).

[10] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler e Sepp Hochreiter. «GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium». In: *arXiv e-prints*, arXiv:1706.08500 (2017).

[11] Irina Higgins et al. «beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework». In: *ICLR*. 2017.

[12] Sepp Hochreiter. «Untersuchungen zu dynamischen neuronalen Netzen». In: 1991.

[13] Danilo Jimenez Rezende, Shakir Mohamed e Daan Wierstra. «Stochastic Backpropagation and Approximate Inference in Deep Generative Models». In: *arXiv e-prints*, arXiv:1401.4082 (2014).

[14] Diederik P Kingma e Max Welling. «Auto-Encoding Variational Bayes». In: *arXiv e-prints*, arXiv:1312.6114 (2013).

[15] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Rapp. tecn. 2009.

[16] Yann Lecun, Léon Bottou, Yoshua Bengio e Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[17] Ziwei Liu, Ping Luo, Xiaogang Wang e Xiaoou Tang. «Deep Learning Face Attributes in the Wild». In: *Proceedings of International Conference on Computer Vision (ICCV)*. 2015.

[18] Tim Salimans et al. «Improved Techniques for Training GANs». In: *arXiv e-prints*, arXiv:1606.03498 (2016).

[19] Kihyuk Sohn, Honglak Lee e Xinchen Yan. «Learning Structured Output Representation using Deep Conditional Generative Models». In: *Advances in Neural Information Processing Systems 28*. A cura di C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama e R. Garnett. Curran Associates, Inc., 2015, pp. 3483–3491.

[20] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens e Zbigniew Wojna. «Rethinking the Inception Architecture for Computer Vision». In: *arXiv e-prints*, arXiv:1512.00567 (2015).

[21] Guido Van Rossum e Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[22] Han Xiao, Kashif Rasul e Roland Vollgraf. «Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms». In: *arXiv e-prints*, arXiv:1708.07747 (2017).