



# UNIVERSITÀ DI PARMA

---

Dipartimento di Ingegneria e Architettura  
Corso di Laurea in Ingegneria Informatica, Elettronica e delle  
Telecomunicazioni

## Trasferimento dell'attenzione in reti GAN cicliche

Attention transfer for cycle consistent generative  
adversarial networks

Relatore:

Chiar.mo Prof. Andrea Prati

Correlatore:

Ing. Tomaso Fontanini

Tesi di Laurea di:  
Filippo Botti matr. 287065

---

ANNO ACCADEMICO 2019-2020

Ai miei genitori.

*“Se ho visto più lontano, è perchè stavo sulle spalle di giganti.”*

*Isaac Newton*

# Ringraziamenti

# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Stato dell'arte</b>	<b>2</b>
1.1 Machine Learning . . . . .	2
1.1.1 Addestramento e discesa del gradiente . . . . .	3
1.2 Deep Learning e Reti Neurali Artificiali . . . . .	5
1.3 Reti Neurali Convolute . . . . .	7
1.4 Generative Adversarial Network . . . . .	10
1.4.1 Pix2Pix Translation . . . . .	12
1.5 GradCam . . . . .	13
1.6 Fréchet Inception Distance . . . . .	15
<b>2 Cycle Gan e GradCam</b>	<b>17</b>
2.1 Image to Image Translation . . . . .	17
2.2 Cycle GAN . . . . .	19
2.3 Codice ed Implementazione . . . . .	22
2.3.1 Networks . . . . .	22
2.3.2 Cycle GAN Model . . . . .	24
2.4 Addestramento . . . . .	26
2.5 GradCam . . . . .	27
2.5.1 Implementazione . . . . .	27
2.5.2 Codice . . . . .	30
2.5.3 CycleGAN e GradCam . . . . .	32

INDICE	iii
<b>3 conclusioni</b>	<b>35</b>
<b>A Appendice</b>	<b>36</b>
<b>Bibliografia</b>	<b>37</b>

# Introduzione

# Capitolo 1

## Stato dell'arte

Illustriamo ora lo stato dell'arte relativo alle Cycle Generative Adversarial Networks (GAN), delle reti di Deep Learning in grado di tradurre immagini da un insieme ad un altro. Verranno analizzate le architetture sulle quali si basano queste reti e i concetti teorici che le caratterizzano. Inoltre verrà trattato un algoritmo per la rappresentazione visiva di come agiscono queste reti: GradCam.

### 1.1 Machine Learning

*“Si dice che un programma apprende dall’esperienza  $E$  con riferimento a alcune classi di compiti  $T$  e con misurazione della performance  $P$ , se le sue performance nel compito  $T$ , come misurato da  $P$ , migliorano con l’esperienza  $E$ .”*

*Tom Mitchell [1]*

Il Machine Learning è una branca dell’intelligenza artificiale che basa il suo funzionamento sulla capacità di apprendere dai dati e dalle esperienze passate [2]. Sostanzialmente il Machine Learning cerca di insegnare ai computer a comportarsi come gli esseri umani: imparando dall’esperienza. Esso infatti non è altro che un metodo di analisi dei dati automatizzato che preso un set



di dati in ingresso è in grado di effettuare una previsione su di esso basandosi su un modello. Per capire meglio il concetto aiutiamoci con un esempio: supponiamo di dover creare un sistema in grado di rilevare la presenza di un cavallo all'interno di un'immagine. Questo compito, svolto da un essere umano dotato di pensiero, è un compito molto semplice: il nostro cervello sa com'è fatta la fisionomia di un cavallo, non deve far altro che confrontare l'immagine che vede con ciò che ricorda ed infine stabilire se è conforme o meno. Ora, supponiamo di voler sostituire l'essere umano con un computer, come facciamo a programmare una macchina per distinguere una foto con un cavallo rispetto a qualsiasi altra foto? L'idea è, appunto, quella di utilizzare un algoritmo di Machine Learning per addestrare il computer a rilevare l'animale. Rimanendo sull'analogia tra essere umano e computer, possiamo sfruttare il ragionamento di base dell'uomo e riportarlo sulla macchina: possiamo mostrare al computer tante immagini di cavalli come esempi, per far sì che impari a riconoscerli. Per insegnargli a riconoscerle dovremo definire un insieme di parametri, detti features, utili al riconoscimento del cavallo. Possiamo, ad esempio, dire al computer di verificare la fisionomia delle orecchie, il colore del manto e la presenza o meno della criniera. Dovremo quindi verificare la presenza di tutti questi tratti e, infine, potremo determinare, con un margine di errore, se la foto in questione ritrae o meno un cavallo.

### 1.1.1 Addestramento e discesa del gradiente

Andando più nel dettaglio, possiamo dire che, dato un insieme in ingresso  $X$  composto dagli esempi e dalle features che il modello deve estrarre e un insieme in uscita  $Y$ , composto dai valori che l'uscita può assumere (1 se presente il cavallo oppure 0, nel nostro caso di esempio), stiamo cercando una funzione  $f(x)$  tale per cui:

$$f(x) : X \rightarrow Y$$

ovvero un'applicazione che associ ai valori  $x$  un'uscita  $y$ .

Per ricavare l'applicazione cercata dovremo associare dei pesi  $w$  alle features,

tali per cui:

$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Dobbiamo quindi trovare il valore ottimo dei pesi in questione e lo facciamo attraverso l'addestramento della rete: partendo con pesi dal valore casuale, per ogni esempio, calcoliamo l'uscita  $y$  e la confrontiamo con il valore associato all'esempio, al fine di calcolare la funzione d'errore (che ci indica, per l'appunto, quanto siamo distanti dal valore corretto). Una volta ottenuta la funzione d'errore possiamo calcolarne il gradiente rispetto ai pesi  $w$ , che ci indicherà la crescita/descrescita della funzione d'errore. Ora, sappiamo quanto vale la funzione d'errore, sappiamo qual è il suo gradiente, possiamo cercare di minimizzarla. Ci scosteremo quindi di un valore pari al *learning rate*, un ulteriore parametro della rete, sulla funzione d'errore, e ripeteremo quanto fatto fino a quando non verrà trovato il valore minimo, che corrisponderà al valore ottimo dei pesi, come vediamo in figura 1.1. Questo approccio viene denominato discesa del gradiente ed è utilizzato per addestrare le reti.

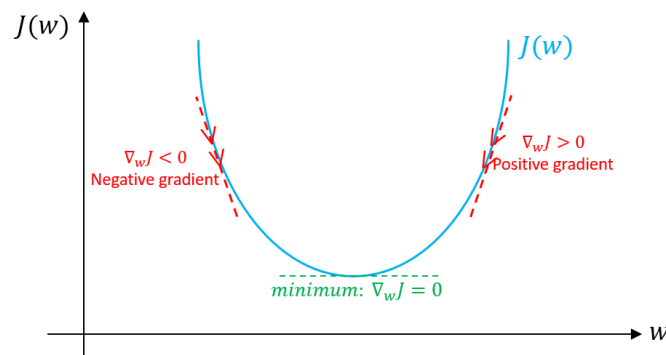


Figura 1.1: Vediamo come data una funzione, attraverso questo approccio si possa trovare il valore minimo.

## 1.2 Deep Learning e Reti Neurali Artificiali

Il Deep Learning è una branca del Machine Learning che mira ad estrarre delle rappresentazioni dei dati partendo dai dati stessi [3]. La differenza sostanziale rispetto al Machine Learning sta nel modo in cui vengono estratte le features: se nel Machine Learning occorre la descrizione di un modello su cui basarsi per estrarle, con il Deep Learning non sarà più necessario. Le reti di Deep Learning infatti sono in grado di estrarre in modo del tutto autonomo queste features. Ovviamente per fare ciò occorre un enorme quantitativo di dati in più rispetto al Machine Learning ed è questo il motivo per il quale il Deep Learning si è sviluppato solo nell'ultimo periodo.

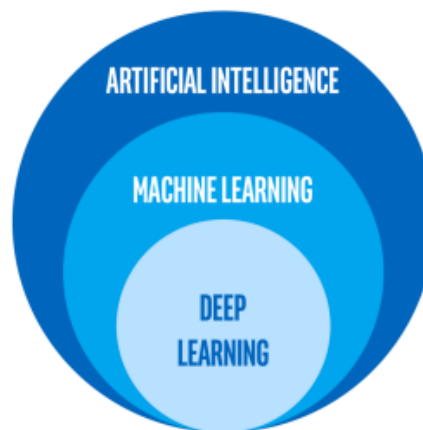


Figura 1.2: Dalla figura capiamo come intelligenza artificiale, Machine Learning e Deep Learning siano strettamente connessi.

Le reti di Deep Learning sono costituite da strati detti *layer*. In un algoritmo di Deep Learning ogni layer ha il compito di estrapolare una caratteristica specifica. Nelle elaborazioni delle immagini, ad esempio, i layer più bassi andranno a definire i bordi e le sfumature dell'immagine, mentre i layer più alti andranno invece a rilevare i concetti rilevanti per l'uomo, come la presenza di un viso o di una lettera. Infine, per concludere questa panoramica iniziale sul Deep Learning occorre specificare cosa si intenda per *hidden layer*,

ovvero i layer nascosti. Gli hidden layer sono i layer interni alla nostra rete neurale, sono quei layer che mettono in comunicazione il primo layer, detto layer di input, con l'ultimo layer, detto layer di output, responsabile della produzione del risultato. Generalmente, per una rete neurale *tradizionale* i layer nascosti sono 2-3, mentre per una rete neurale *profonda*, tipica del Deep Learning, si arriva addirittura a 150 layer nascosti.

Vediamo ora come sono composti questi layer. Dicevamo come il compito di riconoscere un animale in foto fosse molto semplice da un punto di vista umano, mentre molto complesso per una macchina. L'ideale sarebbe avere una macchina dotata di *cervello*, o comunque in grado di emulare il pensiero umano. E' proprio questa l'idea che è alla base delle reti neurali artificiali, ovvero le reti che vanno a comporre gli algoritmi di Deep Learning. Le reti neurali umane sono un insieme di neuroni tra loro connessi, in grado di scambiare informazioni e di attivarsi o meno a seconda delle situazioni [4]. Ricorda qualcosa? Esatto, è la funzione che hanno i layer in una rete di Deep Learning. Una rete neurale artificiale, è, quindi, un'emulazione delle reti neurali umane, composta da un grafo connesso di nodi, come si nota dalla figura 1.3, all'interno del quale viaggiano le informazioni, o meglio i dati.

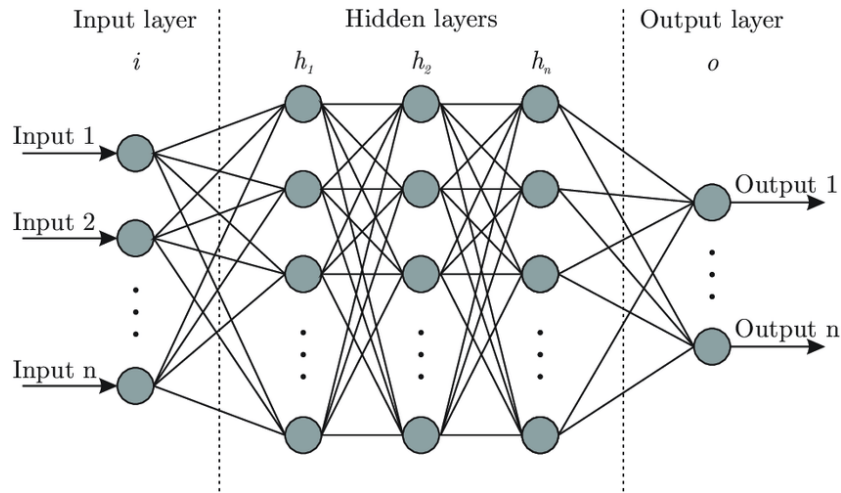


Figura 1.3: Un esempio di rete neurale artificiale, notiamo la suddivisione in layer e come ogni neurone sia connesso ai neuroni del layer successivi

### 1.3 Reti Neurali Convulsive

Tra i tipi di reti neurali le reti neurali convolutive (CNN) sono le reti su cui fare riferimento quando si parla di visione artificiale ed elaborazione delle immagini [5]. Sono infatti reti in grado di catturare con successo ed efficienza le dipendenze spaziali e temporali delle immagini. Sono composte da tre blocchi o livelli: blocco convolutivo, blocco di pooling e i livelli completamente connessi. I primi due blocchi hanno il compito di estrarre le features, mentre il terzo ha il compito di mappare le features estratte nell'output finale.

Per capire il loro funzionamento occorre analizzare come è definita un'immagine. Un'immagine non è altro che un insieme di pixel, ognuno con un determinato valore. Possiamo quindi vedere le immagini come una matrice composta dai valori che assumono i pixel. Le immagini non sono però bidimensionali, infatti le immagini a colori sfruttano i tre canali RGB per comporre i colori, in questo caso avremo quindi una matrice con una dimensione tre di profondità, una per ogni canale. Per intenderci, un'immagine 4K avrà una definizione pari a  $4096 \times 2048$ , un numero quindi elevatissimo

di pixel. E' proprio sulla dimensione che andremo a lavorare con il primo blocco delle reti neurali convolutive: vengono ridotte le dimensioni della matrice attraverso un'operazione di convoluzione, che dà il nome al blocco. Per descrivere il funzionamento definiamo il componente che ci permetterà di lavorare sulla matrice: il kernel, chiamato anche filtro. Il kernel può essere descritto come una matrice di pesi tipicamente di dimensione 3x3:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Il kernel ci permetterà di scorrere tutta l'immagine in altezza ed in larghezza, spostandoci ogni volta di una lunghezza definita a priori, e moltiplicare i valori con i pesi corrispondenti. Successivamente, questi valori verranno tra loro sommati ed andranno a comporre il nuovo valore della nuova matrice risultato, che avrà l'aspetto della figura 1.4.

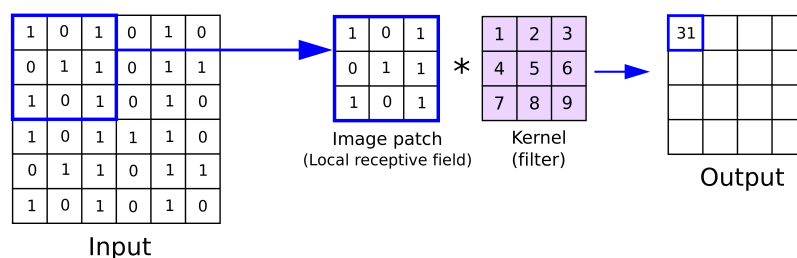


Figura 1.4: Il kernel scorrerà tutta l'immagine al fine di creare una nuova matrice formata dalla somma dei valori dei pixel moltiplicati per i pesi.

Quello che sta realmente accadendo è un'operazione di convoluzione sull'immagine al fine di rilevare caratteristiche come bordi, colore ed orientamento del gradiente. Vi sono due tipi di approcci per questa operazione: il primo approccio, detto *valid padding* permette di ottenere un risultato avente dimensione diversa rispetto alla matrice di partenza, mentre il secondo *same*

*padding* permette, aggiungendo ai margini della matrice iniziale colonne e righe composte da zeri, di ottenere una matrice avente la stessa dimensione di quella di partenza. Durante l'addestramento della rete, lo scopo sarà quello di trovare i valori dei pesi per i quali il kernel fornisce i risultati migliori. Dopo un'operazione di questo tipo, l'output viene fatto passare attraverso una funzione di attivazione, tipicamente una *ReLU* che eliminerà i valori negativi. Il secondo blocco, di pooling, avrà il compito di ridurre le dimensioni ulteriormente, al fine di ridurre la potenza di calcolo necessaria ad elaborare i dati, estrarrà le caratteristiche dominanti ed effettuerà una riduzione del rumore. Il compito di questo blocco sarà quello di ridurre le dimensioni della matrice e raggruppare i valori grazie all'utilizzo di un filtro unitario. Il filtro scorrerà tutta la matrice e, per ogni scorrimento, calcolerà un valore, che sarà il valore che andrà a comporre il risultato. Questo valore può essere calcolato secondo due approcci: *max pooling* dove per ogni raggruppamento il risultato sarà il valore massimo, come in figura 1.5, oppure *average pooling* dove per ogni raggruppamento il risultato sarà la media dei valori.

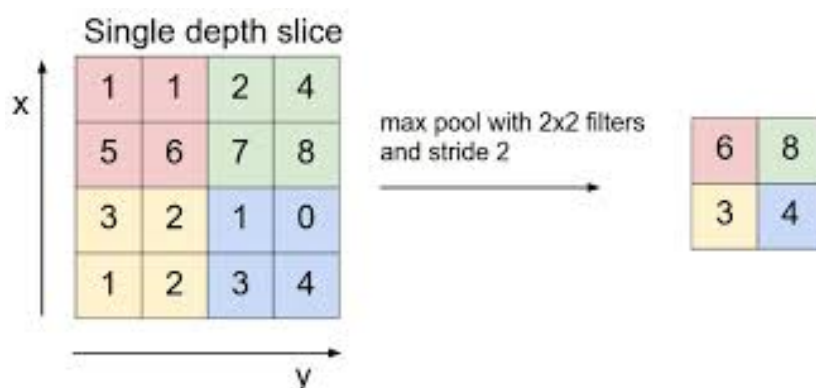


Figura 1.5: In questo caso viene applicato l'approccio max pooling, vediamo come il filtro di dimensione 2x2 ricavi nel primo quadrante della matrice il valore massimo 6 e lo vada ad inserire nella matrice risultato.

Abbiamo visto quindi come vengano catturate le caratteristiche principali della nostra immagine grazie ai primi due blocchi della rete, ora il compito

passerà all'ultimo blocco il quale avrà il compito di convertire i risultati fin'ora ottenuti nella forma adatta agli output finali della rete.

Le CNN sono molteplici, successivamente ci concentreremo sull'architettura ResNet che descrive la rete del nostro progetto.

## 1.4 Generative Adversarial Network

Le Generative Adversarial Network (GAN) sono un modello generativo basato sul Deep Learning [6] utilizzato per la creazione di immagini. Si tratta di un modello basato su due reti: una rete discriminativa, che ha lo scopo di determinare se un'immagine è reale, oppure falsa, ed una rete generativa, il quale compito è appunto quello di generare immagini. Per capire meglio il loro funzionamento aiutiamoci con un paragone: la rete discriminatrice possiamo vederla come un poliziotto, mentre la rete generatrice come un contraffattore. Lo scopo del contraffattore è quello di affinare la propria abilità al fine di produrre opere false e riuscire a spacciarle per vere senza farsi scoprire, mentre lo scopo del poliziotto è quello di ostacolare il contraffattore rilevando se un'opera è reale o meno. La competizione generata da questo duello porta entrambi a migliorarsi, fino a quando non si arriverà al punto in cui il contraffattore riesce a riprodurre opere indistinguibili da quelle reali [7].

Andiamo ora a descrivere queste reti e la teoria sulla quale si basano.

Prima di tutto occorre parlare di una tipologia differente da quella fin'ora trattata di addestramento: l'addestramento non supervisionato. Abbiamo visto come durante la fase di addestramento gli esempi forniti alla rete vengano associati ad un valore di uscita  $y$  e di come la rete si addestri su questi valori. Nell'addestramento non supervisionato, a differenza, agli esempi di addestramento, non vengono associate le uscite  $y$ , ma vengono forniti solo gli input. L'obiettivo di questo addestramento è quello di rendere la rete in grado di rilevare una correlazione tra gli esempi e, di conseguenza, di ricavare un modello. E' ciò che sta alla base dei modelli generativi, ovvero i modelli



in grado di generare nuovi elementi il quanto più possibile simili agli elementi dell'insieme di input.

Dicevamo di come il modello fosse suddiviso in Generatore e Discriminatore, andiamo ora a vedere come lavorano.

Partendo inizialmente da un vettore casuale  $z$ , il generatore  $G$  genererà un'immagine fake  $G(z)$ . Questa immagine sarà poi data in ingresso al Discriminatore  $D$ , che dovrà determinare se l'immagine è reale o falsa, dovrà quindi fare una previsione  $D(G(z))$ . Sul risultato di questa previsione verrà calcolata la funzione d'errore del discriminatore e, con il metodo già ampiamente descritto della discesa del gradiente, verrà addestrato, mantenendo invariati i parametri del Generatore. Sempre con lo stesso metodo verrà successivamente addestrato il Generatore.

Le funzioni d'errore utilizzate nell'addestramento di queste reti sono logaritmiche. Supponiamo il caso di un'immagine reale, i casi possono essere due:

- la previsione è conforme all'immagine, ovvero, determinato con 1 il valore dell'immagine reale, la previsione sarà un valore ad esso vicino, ad esempio  $D(x) = 0.9$ , per cui l'errore commesso sarà piccolo
- la previsione è errata, ad esempio  $D(x) = 0.1$ , per cui l'errore commesso sarà grande

Una funzione che si comporta come quanto descritto è  $f = -\log(D(x))$ , che nel nostro caso avrà come risultato:  $f = 0.1$  per  $D(x) = 0.9$  e  $f = 2.3$  per  $D(x) = 0.1$ . Come visto questa funzione approssima bene il comportamento che vogliamo ottenere dalla funzione d'errore, ovvero è elevata tanto più la previsione è errata.

Con calcoli analoghi, si ottiene come risultato  $f = -\log(1 - D(G(z)))$  nel caso in cui l'immagine sia generata dal Generatore. Quindi, dato che l'obiettivo della rete è quello di generare un'immagine finta indistinguibile da una reale, l'apprendimento consisterà quindi nell'ottimizzare un gioco *minmax* per  $G$  e

$D$ :

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

L'utilizzo delle GAN si concentra su tre campi principali:

- Image Super-Resolution, per migliorare la qualità delle immagini in ingresso;
- Creare arte;
- Image to Image Translation, ovvero l'applicazione di uno stile ad un'immagine di input, come ad esempio estate-inverno, giorno-notte, e, come vedremo più avanti nel nostro caso, cavallo-zebra. Un esempio di questa applicazione è mostrato in figura 1.6.



Figura 1.6: Un esempio di Image to Image translation [8]

### 1.4.1 Pix2Pix Translation

Un aspetto da sottolineare per quanto riguarda le GAN è la possibilità di condizionarne il funzionamento. Parliamo quindi di Conditional Generative Adversarial Network, utilizzate nella traduzione da immagine a immagine Pix2Pix [7].

Abbiamo visto come le GAN generino delle immagini partendo dal rumore. L'idea che è alla base di questa traduzione è quella di partire a generare un'immagine non più da un vettore casuale ottenuto dal rumore, bensì da

un'altra immagine [9]. Aiutandoci con un esempio, supponiamo di avere una GAN utilizzata nella generazione di visi umani. Dalla GAN in questione ci aspettiamo che la probabilità di generare un volto femminile sia pari a quella di un volto maschile, e che quindi la rete non faccia distinzione sull'output generato. Utilizzando una GAN condizionata possiamo, appunto, condizionare la generazione dei volti solo su quelli femminili o solo su quelli maschili. Per fare ciò addestreremo la rete non più su una singola immagine come visto fin'ora, ma su coppie di immagini, grazie ad un dataset *paired*, ovvero un dataset composto da immagini accoppiate a due a due secondo la relazione input-output [8]. In questo modo il discriminatore non lavorerà più facendo previsioni solamente sull'immagine generata, ma lo farà sulla coppia di immagini: immagine di partenza più immagine generata e immagine di partenza più rispettivo output, come si nota dall'esempio in figura 1.7.

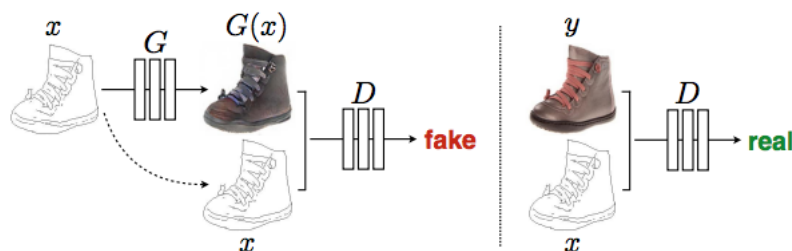


Figura 1.7: Vediamo come il discriminatore esegua una previsione sulla coppia di immagini input-output fake e input-output associato.

## 1.5 GradCam

Abbiamo ampiamente parlato di CNN e di come il loro funzionamento sia ideale o molto performante nell'ambito della visione artificiale e dell'elaborazione delle immagini. Come per ogni ambito dell'intelligenza artificiale, è molto importante capire il loro funzionamento e renderle il più intuitive possibili. Questo perché, nel caso di un malfunzionamento, occorre sapere da che cosa è stato determinato. La necessità di comprendere in maniera

trasparente il loro funzionamento è data da tre principali motivi. In primo luogo, quando l'AI è debole rispetto alla capacità umana è utile identificare il perchè il suo comportamento fallisca. Successivamente quando l'AI è pari alla capacità umana l'obiettivo è quello di stabilire un'adeguata fiducia negli utenti. Infine, quando l'AI è decisamente superiore all'intelligenza umana l'obiettivo è quello di apprendere da essa [10].

Una delle tecniche principali per capire il funzionamento delle CNN è attraverso le mappe di attivazione di classe (CAM). La loro logica è basata sui risultati che produce l'ultimo livello convolutivo di una CNN. Sostanzialmente, grazie alle features map prodotte dall'ultimo layer convolutivo è possibile ricavare dove si focalizza l'attenzione della rete. Per rendere più chiaro il concetto riprendiamo l'esempio della rete classificatrice. Vogliamo capire dove si concentra la rete, su che parametri dell'immagine. Ci aspettiamo che difficilmente l'attenzione sia focalizzata ai margini dell'immagine, o su parametri secondari come lo sfondo, ma che sia focalizzata sulla sagoma dell'animale. CAM ci permette, grazie ad una mappa del calore come vediamo in figura 1.8, di verificare come si comporti la rete nell'analisi dell'immagine.



Figura 1.8: Il risultato dell'applicazione di GradCam su una rete classificatrice. In questo caso l'output di classificazione riguarda il cane e vediamo come l'attenzione sia concentrata tutta sulla faccia dell'animale.

L'approccio visto fin'ora funziona però solo su reti CNN e non su altre ar-

chitetture. Introduciamo quindi GradCam, avente un funzionamento simile a quanto descritto, ma applicabile a qualsiasi tipologia di architettura. Vediamone il funzionamento nel dettaglio. Sia la nostra rete una rete classificatrice e sia  $y^c$  l'output di classificazione della rete. Denotiamo inoltre con  $A^k$  le mappe di attenzione generate dall'ultimo layer convolutivo della rete sulle quali vogliamo andare a calcolare l'attenzione. Calcoliamo il gradiente di  $y^c$  rispetto alle mappe di attenzione  $A^k$ :

$$\frac{\partial y^c}{\partial A^k}$$

Ora, applichiamo un *global average pooling*, ovvero la media su altezza e larghezza di ogni mappa di attenzione, e poniamo:

$$\alpha^c_k = \frac{1}{Z} \sum_{i=1} \sum_{j=1} \frac{\partial y^c}{\partial A^k_{ij}}$$

$\alpha^c_k$  sarà un vettore di dimensione pari al numero di feature map del layer scelto e conterrà i pesi da affidare ad ogni feature map.

Successivamente moltiplicheremo il vettore contenente i pesi con le features map e ne faremo una somma. In questo modo associamo ad ogni canale della convoluzione un peso determinato dal gradiente. Il peso associato a questi canali indica se si attivano o meno durante l'elaborazione dell'immagine. Infine applicheremo una ReLu alla somma ottenuta, al fine di eliminare i valori negativi.

Il risultato ottenuto potrà essere visto attraverso una mappa del calore ed indicherà dove sarà maggiormente focalizzata l'attenzione della rete nel layer stabilito.

## 1.6 Fréchet Inception Distance

Definite le GAN, definito GradCam come metodo per visualizzare dove le reti vanno a lavorare, occorre ora definire un metodo per valutare la qualità della rete generatrice. Introduciamo quindi il Fréchet Inception Distance (FID). Il FID è un metodo utilizzato per valutare la qualità delle immagini

generate da una GAN [11]. Il metodo si basa sul concetto matematico della distanza di Fréchet utilizzato in matematica per calcolare la somiglianza tra le curve tenendo conto della posizione e dell'ordine dei punti lungo le curve [12]. Il ragionamento è quello di valutare due insiemi: un insieme di esempio e un insieme di immagini generate dalla rete. Vengono calcolate quindi le distribuzioni di questi due insiemi, vengono confrontate e viene prodotto un risultato. Il risultato indicherà quanto sono distanti le due distribuzioni, quindi un risultato piccolo è ottimale.

## Capitolo 2

# Cycle Gan e GradCam

In questo capitolo illustreremo l'architettura del nostro progetto e andremo ad analizzare il codice. Partiremo con la definizione del problema della traduzione da immagine a immagine e vedremo com'è composta la rete che abbiamo utilizzato.

### 2.1 Image to Image Translation

(Ho pensato di strutturarla così: breve introduzione sul problema, spiegazione sul perchè pix2pix non sia la soluzione che usiamo, introduzione a cyclegan)

Vi siete mai chiesti come sarebbe bello poter ottenere una foto a colori da una foto in bianco e nero? Oppure avere la possibilità di passare da una foto ritraente un paesaggio estivo a un paesaggio autunnale?

Questi sono solamente alcuni degli esempi di applicazioni della traduzione da immagine a immagine. La traduzione da immagine ad immagine è molto simile a ciò che accade nella traduzione da lingua a lingua: partendo da un'immagine appartenente ad un dominio iniziale, come la foto di un paesaggio estivo, vogliamo ottenere un'immagine appartenente ad un altro dominio, ad esempio passando da paesaggio estivo a paesaggio autunnale.

Un esempio di traduzione da immagine a immagine è il già trattato Pix2Pix

che, grazie ad una GAN condizionata addestrata con un training set paired, di cui un esempio in figura 2.1, produce un output condizionato dall'immagine di input.

Tuttavia, l'approccio Pix2Pix fin qui descritto non è del tutto efficiente in quanto il training set della rete comporta un notevole sforzo nella sua creazione. Infatti, tornando all'esempio dei paesaggi, occorrerebbe avere un enorme quantitativo di foto ritraenti lo stesso identico paesaggio in due differenti momenti dell'anno. In aggiunta, per alcune applicazioni sarebbe del tutto impossibile ottenere un training set di questo tipo: basti pensare alla traduzione da opere d'arte a foto, o all'applicazione di uno stile a immagini reali, come vedremo più avanti nel nostro caso quando vorremo trasformare cavalli in zebre e viceversa.

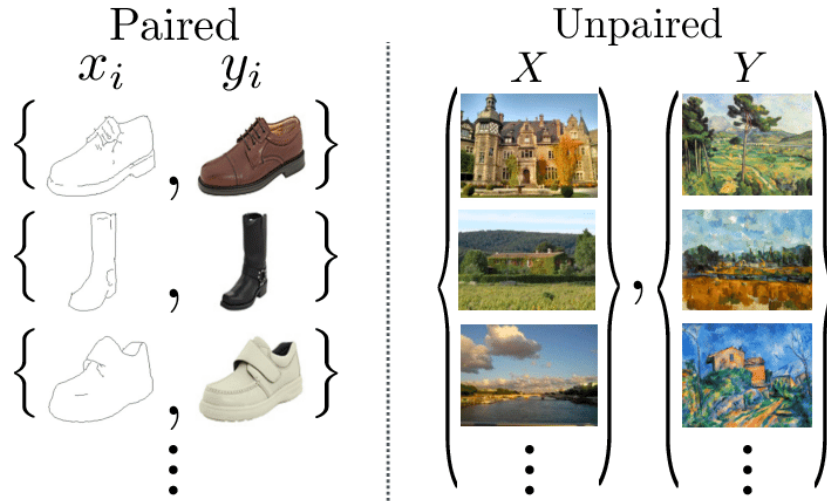


Figura 2.1: Vediamo un esempio di set paired, dove le immagini sono accoppiate tra input-output e set unpaired dove non vi è questa corrispondenza.

Proprio per questo motivo si è pensato di adottare una nuova soluzione attraverso l'addestramento con un training set *unpaired*, dove non vi è una corrispondenza di input-output accoppiati, come vediamo sempre in figura 2.1. In questo caso l'obiettivo è quello di addestrare una rete in grado di mappare le immagini appartenenti al dominio di partenza nel dominio di arrivo,



in modo tale che le immagini generate siano verosimilmente appartenenti al dominio di arrivo.

Tuttavia non siamo ancora arrivati ad un risultato ottimale che soddisfi le nostre esigenze per la traduzione di un'immagine: in questo modo, infatti, non stiamo garantendo la traduzione esatta da immagine di input a immagine di output, ne stiamo solo condizionando la creazione. Inoltre potrebbe subentrare il problema del collasso della rete, dove tutte le immagini di input si mappano nella stessa immagine di output [8]. Occorre quindi trovare un modo per migliorare l'idea fin qui realizzata, e proprio per questo motivo introduciamo una nuova architettura: la Cycle GAN.

## 2.2 Cycle GAN

(Ho pensato di strutturarla così: introduzione, spiegazione matematica del funzionamento della consistenza del ciclo, implementazione per la classe `networks.py` e `cycle-gan-model.py` (solo le funzioni importanti), addestramento) Per spiegare questa architettura possiamo fare un paragone con la linguistica. Quando vogliamo tradurre una frase dall'italiano all'inglese vogliamo che applicando il procedimento di traduzione inversa da inglese a italiano, la frase torni ad essere quella di partenza. Vogliamo cioè che vi sia della consistenza tra i risultati. Questo concetto è ciò proprio che vogliamo applicare al nostro problema della traduzione da immagine a immagine. In poche parole vogliamo ottenere un'immagine di output, la quale applicando a ritroso la traduzione ci restituisca come risultato l'immagine di input.

Matematicamente non vogliamo far altro che trovare due funzioni biettive  $G : X \rightarrow Y$  e  $F : Y \rightarrow X$ , tali per cui  $F$  sia l'inversa di  $G$ , ovvero:

$$\begin{cases} F(G(x)) = x \\ G(F(y)) = y \end{cases}$$

Da un punto di vista architetturale questo problema può essere risolto attraverso due reti GAN, la prima responsabile della traduzione da  $X$  a  $Y$ , la

seconda responsabile della traduzione inversa da  $Y$  a  $X$  [8]. In questo modo il generatore della prima rete prenderà in input le immagini dal primo dominio e produrrà in output le immagini per il secondo dominio, successivamente il compito passerà al generatore della seconda rete che prenderà come input le immagini generate dal primo generatore e produrrà come output un'immagine del primo dominio il più fedele possibile, poi vedremo come, all'immagine di partenza. I rispettivi discriminatori saranno impiegati nel determinare quanto le immagini generate siano plausibili.

Per quanto riguarda l'addestramento, esso è conforme a quanto già descritto per le reti GAN, con l'aggiunta di una loss che indica la consistenza del ciclo. Per spiegare come viene calcolata questa loss aggiuntiva definiamo due concetti: *forward cycle consistency* e *backward cycle consistency*. Il forward cycle consistency (consistenza del ciclo in avanti) non è altro che la sopra definita  $F(G(x)) = x$ , mentre il backward cycle consistency (consistenza del ciclo all'indietro) è la sopra definita  $G(F(Y)) = y$ , come si può intuire visivamente dalla figura 2.2. Attraverso questi due concetti possiamo quindi valutare di quanto si discosta l'immagine rigenerata, grazie al forward cycle consistency, con l'immagine iniziale. Infatti, calcolando la differenza (che matematicamente rappresentiamo con la norma, o la norma al quadrato) tra le due immagini si ottiene l'errore che la rete commette durante la traduzione dell'immagine. Possiamo applicare il medesimo ragionamento per quanto riguarda il backward cycle consistency per ottenere la loss di consistenza del ciclo:

$$L_{cyc}(G, F) = ||G(F(x)) - x|| + ||F(G(Y)) - y||$$

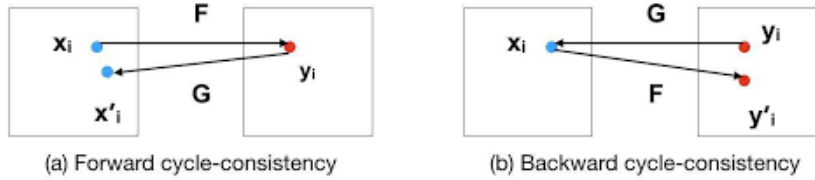


Figura 2.2: Vediamo una rappresentazione grafica della consistenza del ciclo.

In questo modo abbiamo quindi definito una rete costituita da due GAN cicliche, da cui il nome. La loss totale della rete dipenderà quindi dalle loss delle due GAN, più la loss aggiuntiva che indica la consistenza del ciclo.

$$L_{\text{cycgan}}(G, F, D_x, D_y) = L_{\text{gan}}(G, D_y, X, Y) + L_{\text{gan}}(F, D_x, Y, X) + L_{\text{cyc}}(G, F)$$

Come ampiamente spiegato la Cycle GAN può essere definita attraverso due funzioni biettive, una l'inversa dell'altra. Un problema di traduzione da immagine a immagine risolto attraverso l'implementazione di una rete di questo tipo porterà quindi ad una doppia soluzione: la traduzione, che possiamo definire *primaria*, da dominio  $A$  a dominio  $B$  ed una traduzione *secondaria* da dominio  $B$  a dominio  $A$ . Vedremo infatti nei capitoli futuri come il nostro progetto si concentri sulla traduzione di immagini da cavalli a zebre, ma noteremo come, seppur con risultati inferiori, la rete sia in grado di tradurre anche da zebre a cavalli, come vediamo tra gli esempi di figura 2.3.

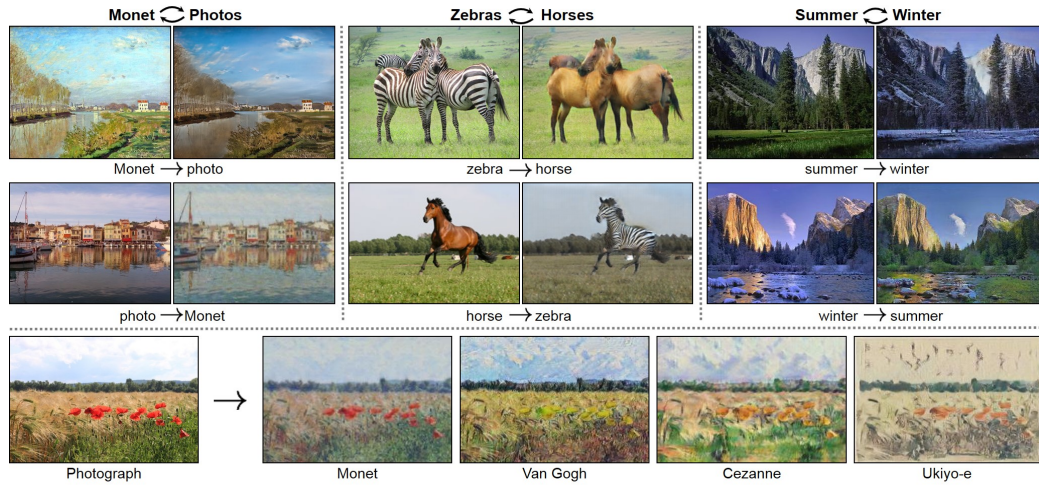


Figura 2.3: Vediamo vari esempi di una image to image translation attraverso l'uso di Cycle GAN.

## 2.3 Codice ed Implementazione

Dopo aver trattato la teoria sulla quale si basano le Cycle GAN vediamo in questa sezione com'è possibile implementarle a livello di codice. L'implementazione iniziale da cui partiremo è quella proposta dal Paper [8].

Il codice è scritto in Python e utilizza il Framework Pytorch per costruire la rete e addestrarla. Il codice dà la possibilità di specificare diversi parametri utili alla rete, in questa sezione ci limiteremo a definire quelli utilizzati nel nostro progetto.

### 2.3.1 Networks

La prima classe che andiamo a definire è la classe networks, all'interno della quale vengono definite le architetture utilizzate per implementare la CycleGAN. Come prima cosa andremo a definire le reti Generatrici, che nel nostro progetto saranno composte attraverso un'architettura ResNet avente nove blocchi residui. Notiamo come durante l'inizializzazione del Generatore sia possibile andare a specificare alcuni parametri, per cambiare il comportamento della rete. Tra questi troviamo la tipologia dell'architettura utilizzata, la quale, nel nostro caso, sarà, come detto, una ResNet.

```
def define_G(input_nc, output_nc, ngf, netG, norm='batch',
             use_dropout=False, init_type='normal',
             init_gain=0.02, gpu_ids=[]):

    norm_layer = get_norm_layer(norm_type=norm)

    if netG == 'resnet_9blocks':
        net = ResnetGenerator(input_nc, output_nc, ngf,
                              norm_layer=norm_layer,
                              use_dropout=use_dropout,
                              n_blocks=9)
```

Con codice analogo vengono successivamente definite le reti Discriminatrici. Anche in questo caso sarebbe possibile specificare ulteriori parametri per cambiare il tipo di rete utilizzato. Si distinguono tre tipologie di Discriminatore: *basic*, *n-layer* e *pixel*. Nella nostra rete utilizzeremo il discriminatore Basic, il quale suddivide l'immagine in parti composte da 70x70 pixel e per ognuna di queste viene effettuata una classificazione tra reale e fake. Successivamente viene definita la Loss della GAN, nel nostro progetto utilizzeremo una MSELoss.

```
def __init__(self, gan_mode, target_real_label=1.0,
              target_fake_label=0.0):
    if gan_mode == 'lsgan':
        self.loss = nn.MSELoss()
```

Proseguendo all'interno del codice vengono definiti i blocchi residui della nostra ResNet. Come detto in precedenza la nostra rete sarà composta da nove blocchi residui. I blocchi residui vengono utilizzati nel Deep Learning al fine di poter ottenere una maggiore profondità della rete. Infatti, se anzichè utilizzare una ResNet utilizzassimo un'altra architettura, sarebbe errato parlare di precisione della rete proporzionale alla sua profondità, in quanto subentrerebbe il problema del *gradiente di fuga*, secondo il quale il gradiente si annullerà a causa dei troppi livelli. Per questo motivo vengono introdotte le ResNet, le quali attraverso *salti* tra i layer che compongono i blocchi residui, riescono ad evitare questo problema [13].

Ogni blocco residuo è composto da i seguenti layer:

```
ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d
      (1): Conv2d
      (2): InstanceNorm2d
      (3): ReLU
      (4): ReflectionPad2d
```

```
(5): Conv2d
(6): InstanceNorm2d
)
```

### 2.3.2 Cycle GAN Model

Dopo aver descritto come viene definita l'architettura della rete, possiamo descrivere come la Cycle GAN è implementata.

Come prima cosa occorre ovviamente costruire i due generatori e i due discriminatori, richiamando i sopra citati metodi della classe networks:

```
self.netG_A = networks.define_G(opt.input_nc, opt.output_nc,
                                opt.ngf, opt.netG, opt.norm,
                                not opt.no_dropout, opt.init_type,
                                opt.init_gain, self.gpu_ids)
self.netG_B = networks.define_G(opt.output_nc, opt.input_nc,
                                opt.ngf, opt.netG, opt.norm,
                                not opt.no_dropout, opt.init_type,
                                opt.init_gain, self.gpu_ids)
self.netD_A = networks.define_D(opt.output_nc, opt.ndf,
                                opt.netD, opt.n_layers_D, opt.norm,
                                opt.init_type, opt.init_gain,
                                self.gpu_ids)
self.netD_B = networks.define_D(opt.input_nc, opt.ndf,
                                opt.netD, opt.n_layers_D, opt.norm,
                                opt.init_type, opt.init_gain,
                                self.gpu_ids)
```

Successivamente definiamo il comportamento dei Generatori che avranno il compito di generare le immagini fake attraverso le quali andremo a calcolare la Loss:

```
def forward(self):
    self.fake_B = self.netG_A(self.real_A)  #  $G_A(A)$ 
    self.rec_A = self.netG_B(self.fake_B)   #  $G_B(G_A(A))$ 
    self.fake_A = self.netG_B(self.real_B)  #  $G_B(B)$ 
    self.rec_B = self.netG_A(self.fake_A)   #  $G_A(G_B(B))$ 
```

Come vediamo le istruzioni rappresentano il funzionamento descritto per le Cycle GAN: partendo da un'immagine del dominio  $A$ , generiamo con il primo generatore un'immagine fake per il dominio  $B$ , successivamente attraverso il secondo generatore ricostruiamo l'immagine iniziale partendo dall'immagine fake del dominio  $B$ . Questo procedimento è ripetuto in maniera analoga anche per la generazione da dominio  $B$  a dominio  $A$ .

Seguentemente definiamo la funzione Backward per i due discriminatori, alla quale passiamo come argomenti le immagini reali e le immagini fake, su cui i discriminatori dovranno effettuare la previsione, al fine di calcolare la Loss dei discriminatori.

```
def backward_D_basic(self, netD, real, fake):
    pred_real = netD(real)
    loss_D_real = self.criterionGAN(pred_real, True)
    # Fake
    pred_fake = netD(fake.detach())
    loss_D_fake = self.criterionGAN(pred_fake, False)
    # Combined loss and calculate gradients
    loss_D = (loss_D_real + loss_D_fake) * 0.5
    loss_D.backward()
```

Infine definiamo il metodo di Backward per quanto riguarda i generatori: è qui che calcoleremo la Loss di consistenza del ciclo.

```
# Forward cycle loss ||  $G_B(G_A(A)) - A$  ||
self.loss_cycle_A = self.criterionCycle(self.rec_A, self.real_A)
```

```
# Backward cycle loss || G_A(G_B(B)) - B||  
self.loss_cycle_B = self.criterionCycle(self.rec_B, self.real_B)
```

Oltre alla Loss di consistenza del ciclo sono ovviamente presenti le classiche Loss per le reti GAN e una Loss di identità ottenuta passando ad un generatore un'immagine del dominio di arrivo, teoricamente il generatore in questione dovrebbe produrre come output l'immagine stessa in quanto già appartenente al dominio d'arrivo.

## 2.4 Addestramento

Abbiamo quindi definito tutti i punti più importanti delle classi che definiscono la nostra rete, ora possiamo concentrarci sull'addestramento. Per prima cosa occorre definire un dataset sul quale andare a lavorare. Il progetto dal quale siamo partiti mette a disposizione svariati dataset su cui addestrare la rete, tra questi il dataset *horse2zebra*. Il dataset in questione offre la possibilità di tradurre immagini di cavalli in zebre, quindi applicando le strisce bianche e nere al manto del cavallo, e da zebre a cavalli, quindi applicando il procedimento inverso rimuovendo le strisce bianche e nere e dando un colore omogeneo al manto.

Per l'addestramento della rete si è scelto di utilizzare Google Colab, un ambiente gratuito offerto dalla Google che mette a disposizione le proprie schede grafiche per addestrare reti di questo tipo.

L'addestramento si basa su epoche e per la nostra rete il numero di epoche fissato è pari a 200.

Per quanto riguarda i tempi di addestramento essi ovviamente variano a seconda della rete che si vuole addestrare. Nel nostro caso i tempi impiegati dalla Cycle GAN fin qui descritta erano dell'ordine dei minuti per epoca, indicativamente dieci. Per le modifiche successive che apporteremo alla rete vedremo come il tempo necessario all'addestramento aumenterà, fino ad arrivare a oltre quaranta minuti per epoca per la Cycle GAN addestrata grazie all'utilizzo di GradCam su quattro blocchi residui, che descriveremo nei ca-



pitoli successivi.

(parlo qui delle modifiche che ho fatto io al codice, quindi aggiungendo una sezione contenente la spiegazione di GradCam.py e di come collabori con cycle-gan-model.py, oppure lo faccio nel prossimo capitolo?)

## 2.5 GradCam

Nel capitolo precedente abbiamo trattato l'algoritmo GradCam definendolo come un algoritmo in grado di associare una risposta visiva al comportamento della rete, indicandoci attraverso una mappa di calore dove la rete pone la propria attenzione durante la generazione delle immagini. L'idea è quindi quella di sfruttare GradCam al fine di indirizzare l'attenzione della rete durante la generazione delle immagini. Idealmente, infatti, le mappe di attenzione durante la ricostruzione dell'immagine iniziale, quindi dal dominio  $B$  al dominio  $A$ , dovrebbero essere uguali a quelle della trasformazione da  $A$  a  $B$ . Infatti, tornando all'esempio del dataset cavalli e zebre, ci aspettiamo che se la mappa di attenzione generata dalla rete durante la trasformazione da cavallo a zebra si focalizza sulla sagoma del cavallo per inserirne le strisce bianche e nere, anche l'attenzione generata dalla ricostruzione da zebra fake a cavallo si focalizzi negli stessi punti della precedente, in modo tale da rimuovere le strisce aggiunte e riportare il manto del cavallo allo stato iniziale. Il ragionamento è molto simile alla già illustrata consistenza del ciclo per le CycleGAN: deve esserci consistenza tra le mappe di attenzione. Lo scopo del nostro progetto è dunque quello di introdurre una nuova loss che indichi questa uguaglianza.

### 2.5.1 Implementazione

Vediamo ora come applicare il ragionamento fin qui spiegato. Come prima cosa occorre definire i parametri su cui lavora GradCam. Come già accennato nei capitoli precedenti, GradCam lavorerà sui layer di una rete, quindi occorrerà scegliere tra i layer che compongono la nostra rete quelli sui quali

andare a generare le mappe di attenzione. Oltre a questo, occorre andare a definire qual è l'output di classificazione su cui effettuare la backpropagation. Nel nostro caso non abbiamo un vero e proprio output di classificazione, ma possiamo sfruttare l'output del Discriminatore come tale. Infatti il comportamento del Discriminatore è molto simile ad un classificatore: possiamo vedere la determinazione tra immagine vera o immagine fake come una vera e propria classificazione.

Abbiamo quindi definito tutti i parametri fondamentali per implementare GradCam in una CycleGAN, possiamo concentrarci ora sull'analisi delle istruzioni fondamentali. Prima di procedere con l'analisi del codice, però, dobbiamo definire su quali layer ricavare le mappe di attenzione da utilizzare per il calcolo della Loss sopra citata. La rete generatrice, infatti, è composta da 23 blocchi sui quali è possibile andare a generare le mappe di attenzione. Idealmente, si potrebbero utilizzare tutti e 23 i layer, magari associando un peso diverso per ognuno di essi, ma non è una strada ottimale, in quanto non tutti i layer sono utili al nostro scopo. La scelta dei layer su cui calcolare la Loss ricade quindi sui nove blocchi residui della rete, in quanto come vediamo in figura 2.4, sono i layer che offrono i risultati migliori. Come vedremo successivamente, i risultati migliori sono quelli ottenuti utilizzando solo l'ultimo blocco residuo per calcolare la Loss di consistenza dell'attenzione.

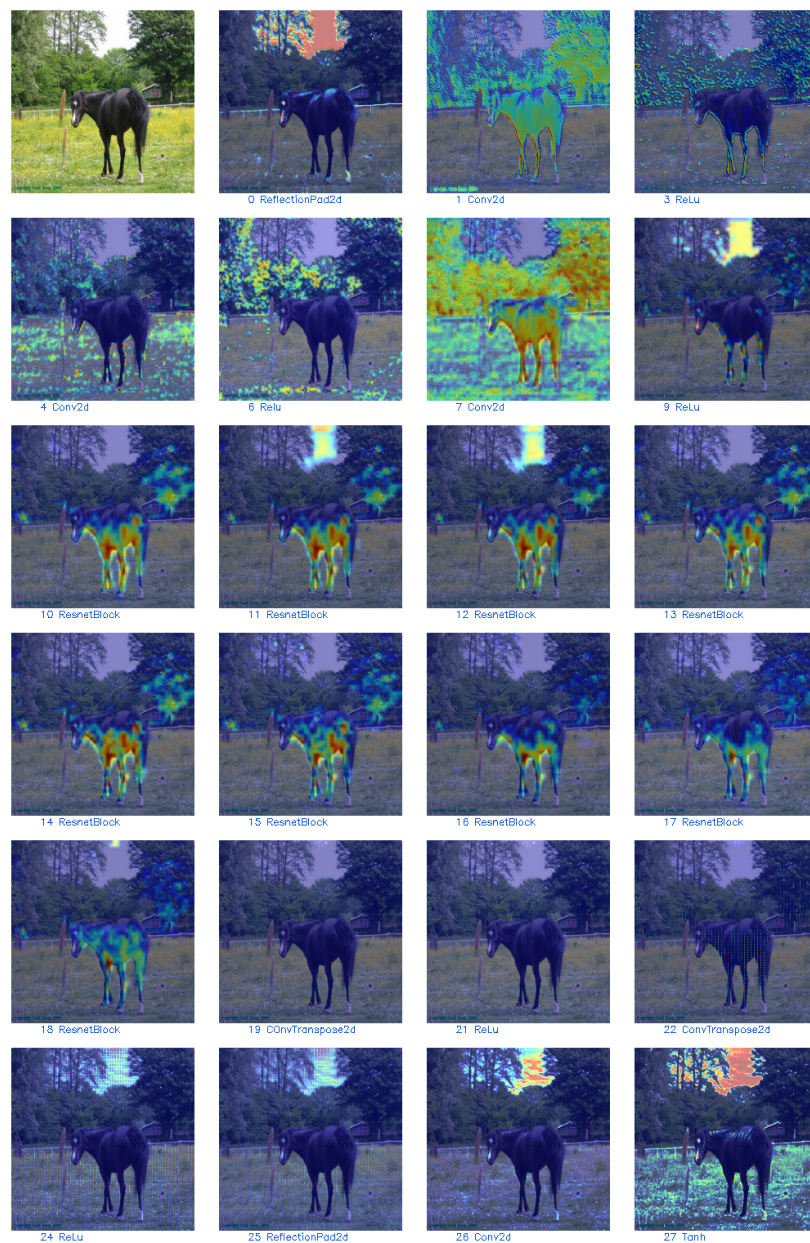


Figura 2.4: Vediamo le varie mappe di attenzione generate applicando GradCam su tutti i blocchi possibili della rete.

### 2.5.2 Codice

Dopo aver definito tutti i parametri sui quali lavorare possiamo soffermarci sul codice necessario a realizzare quanto spiegato fin'ora. Definiamo in primo luogo la classe `FeatureExtractor`, la quale ha il compito di salvare il gradiente dei layer scelti per generare le mappe di attenzione.

```
def __init__(self, model, target_layers):
    self.model = model
    self.target_layers = target_layers
    self.gradients = []
def __call__(self, x):
    outputs = []
    self.gradients = []
    for name, module in self.model._modules.items():
        x = module(x)
        if name in self.target_layers:
            x.register_hook(self.save_gradient)
            outputs += [x]
    return outputs, x
```

Come vediamo scorriamo tutti i layer della rete e li applichiamo all'immagine  $x$ . Successivamente verifichiamo se il layer in questione è tra quelli sui quali vogliamo generare le mappe di attenzione: in caso affermativo salviamo il gradiente e ne salviamo l'output in uscita dal Generatore, precedentemente calcolato, in una lista.

Successivamente definiamo la classe `ModelOutputs`, che ha il compito di sfruttare l'output del Discriminatore per generare le mappe di attenzione.

```
class ModelOutputs():
    def __init__(self, model, discriminator, feature_module,
                  target_layers):
        self.model = model
```

```

        self.feature_module = feature_module
        self.discriminator = discriminator
        self.feature_extractor =
            FeatureExtractor(self.feature_module,
                            target_layers)

    def get_gradients(self):
        return self.feature_extractor.gradients

    def __call__(self, x):
        target_activations = []
        for name, module in self.model._modules.items():
            if module.model == self.feature_module:
                target_act, x = self.feature_extractor(x)
            elif "avgpool" in name.lower():
                x = module(x)
                x = x.view(x.size(0), -1)
            else:
                x = module(x)
                x = self.discriminator(x)
        return target_act, x

```

Vediamo come si scorrono tutti i layer del Generatore e per ognuno di essi si verifici se è tra quelli su cui vogliamo generare l'attenzione: in caso affermativo, viene salvato il gradiente, grazie alla classe `FeatureExtractor` sopra citata. Infine, per ogni layer ne viene calcolato l'output, generando quindi l'immagine fake. L'immagine generata viene di conseguenza passata al discriminatore, al fine di determinare l'output di classificazione necessario a GradCam per generare le mappe di attenzione. Infine, definiamo la classe `GradCam` avente il compito di generare, grazie all'utilizzo delle classi precedenti, le mappe di attenzione. Dopo aver richiamato il metodo dell'estrazione delle features, determiniamo un tensore *one\_hot* contenente la media dei valo-

ri di classificazione determinati dal Discriminatore su cui andiamo a fare la backpropagation. Successivamente, andiamo a definire il tensore *grads\_val*, attraverso il quale andiamo a recuperare i gradienti precedentemente salvati. Infine, tramite questi gradienti, viene definita la mappa di attenzione *cam* cercata.

```
features, output = self.extractor(input)
one_hot = torch.mean(output)

one_hot.backward(retain_graph=True)

grads_val = self.extractor.get_gradients()[-1].cpu().data

target = features[-1]
target = target.cpu().data[0, :]

weights = torch.mean(grads_val, axis=(2, 3))[0, :]
cam = torch.zeros(target.shape[1:], dtype=torch.float32)

for i, w in enumerate(weights):
    cam += w * target[i, :, :]

cam = F.relu(cam)
cam = cam - torch.min(cam)
cam = cam / torch.max(cam)
return cam
```

### 2.5.3 CycleGAN e GradCam

Ora, come ultimo passaggio non ci resta che combinare insieme le due architetture fin qui definite, al fine di poter calcolare la Loss di consistenza dell'attenzione. Per fare questo quindi generiamo le mappe di attenzione al-

l'interno della classe cycleGan model sia sulla traduzione da  $A$  a  $B$  che nella ricostruzione da  $B$  a  $A$ .

```
def __init__(self, opt):
    self.gradcamG_A = GradCam(model=self.netG_A,
                               discriminator=self.netD_A,
                               feature_module=self.netG_A.module.model,
                               use_cuda=True)
    self.gradcamG_B = GradCam(model=self.netG_B,
                               discriminator=self.netD_B,
                               feature_module=self.netG_B.module.model,
                               use_cuda=True)

def forward(self):
    self.fake_B = self.netG_A(self.real_A)
    horse = self.real_A.requires_grad_(True)
    self.cam_fake_B = self.gradcamG_A(horse, self.first_layer,
                                       None)

    self.cam_fake_B = self.cam_fake_B.unsqueeze(0)
    for name in (self.layers):
        self.cam_fake_B = torch.cat((self.gradcamG_A(horse,
                                                       name, None).unsqueeze(0),
                                     self.cam_fake_B), dim=0)
```

Come vediamo dal codice, generiamo le mappe di attenzione sull'immagine reale per ogni layer tra quelli da noi indicati. Questa operazione viene ripetuta, ovviamente con i rispettivi generatori e discriminatori, anche per l'immagine ricostruita e per la traduzione inversa da zebre a cavalli.

Concludiamo ora la parte relativa al codice andando ad implementare la Loss di consistenza dell'attenzione. Per calcolarla utilizziamo una *MSELoss*, ovvero la norma al quadrato tra le due attenzioni generate.

```
def backward_G(self):  
    self.att_lossA2B = self.MSE_LOSS(self.cam_fake_B.detach(),  
                                       self.cam_rec_A)  
    self.att_lossB2A = self.MSE_LOSS(self.cam_fake_A.detach(),  
                                       self.cam_rec_B)
```

Analizzeremo successivamente il perchè vi sia un *detach* sulle mappe di attenzione delle immagine fake.



## Capitolo 3

### conclusioni

Appendice A

Appendice

# Bibliografia

- [1] Mitchell Tom. Hill, mcgraw. *Machine Learning*, 1997.
- [2] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.
- [3] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [4] Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. *Elements of artificial neural networks*. MIT press, 1997.
- [5] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4):611–629, 2018.
- [6] Jason Brownlee. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation*. Machine Learning Mastery, 2019.
- [7] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.
- [8] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

- 
- [9] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.
  - [10] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
  - [11] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a nash equilibrium. *CoRR*, abs/1706.08500, 2017.
  - [12] Thomas Eiter and Heikki Mannila. Computing discrete fréchet distance. Technical report, Citeseer, 1994.
  - [13] Sasha Targ, Diogo Almeida, and Kevin Lyman. Resnet in resnet: Generalizing residual architectures. *arXiv preprint arXiv:1603.08029*, 2016.
  - [14] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.
  - [15] W. J. Zhang, G. Yang, Y. Lin, C. Ji, and M. M. Gupta. On definition of deep learning. In *2018 World Automation Congress (WAC)*, pages 1–5, 2018.