



UNIVERSITÀ  
DEGLI STUDI  
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea  
in INGEGNERIA INFORMATICA

Elaborato per il corso di  
DEEP LEARNING  
ANIMEGAN

**Professore:** Ivan Serina

715198 Fabio Fiorini  
717051 Axel Mastroianni

---

Anno Accademico 2021/2022

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Descrizione Del Problema . . . . .	1
1.2	Affrontare il problema . . . . .	2
1.3	Repository . . . . .	3
<b>2</b>	<b>Preparazione Dataset</b>	<b>4</b>
2.1	Cercare le Immagini . . . . .	4
2.2	Esplorazione Di Getchu . . . . .	4
2.3	Rifattorizzazione dello Scraper . . . . .	5
2.4	Organizzazione Dei File . . . . .	8
2.5	Rimuovere Le Non Ragazze . . . . .	9
2.6	Face Detection e Extraction . . . . .	11
<b>3</b>	<b>GAN</b>	<b>13</b>
3.1	Cosa Sono le GAN . . . . .	13
3.1.1	Generator . . . . .	13
3.1.2	Discriminator . . . . .	15
3.2	Le GAN: Un gioco a due giocatori . . . . .	16
3.3	Le GAN e le Reti Convoluzionali . . . . .	17
3.3.1	Upsampling . . . . .	17
3.3.2	Convoluzione Trasposta . . . . .	18
3.4	Algoritmo Di Training Delle GAN . . . . .	19
3.4.1	GANs Loss Function . . . . .	20
<b>4</b>	<b>DCGAN</b>	<b>22</b>
4.1	Implementazione . . . . .	23
4.2	Risultati . . . . .	28

<b>5 WGAN</b>	<b>37</b>
5.1 Implementazione . . . . .	38
5.2 Risultati . . . . .	41
<b>6 WGAN-GP</b>	<b>45</b>
6.1 Wassestein GAN . . . . .	45
6.2 Utilizzo della Penalità Del Gradiente . . . . .	46
6.3 Implementazione . . . . .	48
6.4 Risultati . . . . .	51
<b>Conclusioni e sviluppi futuri</b>	<b>55</b>
<b>7 Conclusioni e sviluppi futuri</b>	<b>55</b>
<b>Bibliografia</b>	<b>56</b>

# Elenco delle figure

1.1	Esempio di immagine . . . . .	1
1.2	I risultati del paper . . . . .	2
2.1	Home Page di Getchu . . . . .	4
2.2	Esempio di Personaggi . . . . .	5
2.3	Parte finale dello scraper . . . . .	7
2.4	Estrazione Ragazze-NonRagazze . . . . .	10
2.5	Esempio di rettangolo originale (blu) vs ingrandito di 1.5 volte (rosso) . .	12
2.6	Sulla destra il volto estratto con il rettangolo rosso, sulla sinistra quello con il rettangolo blu . . . . .	12
3.1	Modello del Generator . . . . .	14
3.2	Modello del Discriminator . . . . .	15
3.3	Gioco a due Giocatori a somma zero . . . . .	16
3.4	Esempio di Upsampling . . . . .	17
3.5	Matrice di Partenza . . . . .	18
3.6	Matrice dopo La Convoluzione Traposta con stride=2 e kernel=1 . . . . .	19
3.7	Pseudo Codice Algoritmo Di Training . . . . .	19
4.1	Discriminatore . . . . .	23
4.2	Generatore . . . . .	24
4.3	Script per effettuare un nuovo training oppure ripartire da un modello già allenato . . . . .	25
4.4	Inizializzazione dei pesi . . . . .	26
4.5	Fase di training . . . . .	27
4.6	Risultato dopo 1 epoca . . . . .	29
4.7	Risultato dopo 10 epoche . . . . .	30
4.8	Risultato dopo 100 epoche . . . . .	31

4.9	Risultato dopo 200 epoche . . . . .	32
4.10	Risultato dopo 400 epoche . . . . .	33
4.11	Risultato dopo circa 800 epoche . . . . .	34
4.12	Grafico delle loss function per le epoche dalla 50 alla 100 . . . . .	35
4.13	Grafico delle loss function per le epoche dalla 165 alla 200 . . . . .	35
4.14	Grafico delle loss function per le epoche dalla 375 alla 400 . . . . .	36
4.15	Grafico delle loss function per le epoche dalla 775 alla 800 . . . . .	36
5.1	Algoritmo di training della WGAN . . . . .	37
5.2	Generator . . . . .	38
5.3	Discriminator . . . . .	39
5.4	Training . . . . .	40
5.5	Risultato dopo 10 epoche . . . . .	41
5.6	Risultato dopo 100 epoche . . . . .	42
5.7	Risultato dopo 200 epoche . . . . .	43
5.8	Grafico delle loss function per le epoche dalla 0 alla 50 . . . . .	44
5.9	Grafico delle loss function per le epoche dalla 150 alla 200 . . . . .	44
6.1	Earth Move Distance o Wasserstein Loss . . . . .	45
6.2	Divergenze con Weight Clipping Rispetto a Gradient Penalty . . . . .	46
6.3	WGAN-GP Loss . . . . .	47
6.4	Algoritmo di Training della WGAN-GP . . . . .	47
6.5	Generator . . . . .	48
6.6	Discriminator . . . . .	49
6.7	Training . . . . .	49
6.8	Algoritmo di Gradient Penalty . . . . .	50
6.9	Risultati dopo 10 epoche . . . . .	51
6.10	Risultati dopo 100 epoche . . . . .	52
6.11	Risultati dopo 200 epoche . . . . .	52
6.12	Risutati dopo 400 epoche . . . . .	53
6.13	Andamento generale della loss . . . . .	53

# 1. Introduzione

## 1.1 Descrizione Del Problema

Il problema che abbiamo deciso di affrontare per questo progetto è quello di esplorare il mondo delle Generative Adversarial Networks (GANs) generando facce di personaggi di visual novel come ad esempio quella che viene riportata nella figura 1.1



Figura 1.1: Esempio di immagine

Non c’è una reale e convincente motivazione riguardo l’utilità del progetto se non evitare a chi volesse trarre spunti per la creazione di personaggi la seccatura di cercare su internet ispirazione ma utilizzare un modello generativo per avere personaggi sicuramente originali

Un’altra motivazione è sicuramente quella di voler vedere in azione le GAN al di fuori dei soliti esempi sui dataset di MNIST o CIFAR e invece creare noi un dataset passando quindi attraverso tutti gli step che lo rendono adatto ad essere utilizzato per addestrare un modello di Deep Learning.

Ultima motivazione, ma non meno importante, risiede nell’aver trovato un paper che riguardava proprio un progetto molto simile e quindi è nata in noi la curiosità di provare ad emularne i risultati utilizzando architetture diverse da quelle presentate dall’autore

In figura 1.2 possiamo vedere i risultati conseguiti nel paper di riferimento. Gli autori hanno effettuato il training su più di 52k immagini.



Figure 7: Generated samples

Figura 1.2: I risultati del paper

## 1.2 Affrontare il problema

Siccome il nostro progetto è alquanto originale rispetto a quello che si trova comunemente su internet cercando tutorial sulle GAN abbiamo innanzitutto dovuto affrontare il problema di costruire un nostro dataset e successivamente anche quello di ricercare e costruire una GAN che andasse bene.

Per il dataset abbiamo dovuto quindi prima raccogliere le immagini dal web, da un sito chiamato [getchu](#), sono state rimosse quelle ritenute non opportune per il dataset, abbiamo

poi esplorate per capirne le caratteristiche e infine abbiamo costruito 3 architetture GAN:  
DCGAN, WGAN e WGAN-GP

Avremmo voluto anche provare la StyleGAN, tuttavia le risorse computazionali di Colab,  
dei nostri computer o della macchina condivisa con noi dall'università non ne permetteva-  
no il training in tempi ragionevoli o addirittura non lo permettevano affatto

### 1.3 Repository

Qualora siate interessati al codice dei modelli li potete trovare nella nostra repository  
[GitHub](#)

## 2. Preparazione Dataset

### 2.1 Cercare le Immagini

Come abbiamo accennato non esiste nessun dataset pre-costruito per il nostro problema ma è stato necessario costruirlo, cercando su internet numerose immagini

La nostra fortuna è stata trovare una nota a piè di pagina sul paper a cui ci siamo ispirati che indicava che la fonte di provenienza delle immagini era un sito chiamato [getchu](#)

### 2.2 Esplorazione Di Getchu

Ad un primo sguardo questo sito web ci è sembrato completamente inutile siccome era tutto in Giapponese e inoltre sulla sua home page era presente di tutto fuorché un sacco di immagini



Figura 2.1: Home Page di Getchu

Esplorandolo un po' abbiamo innanzitutto capito che il sito vendeva Visual Novel in stile anime e ognuna di esse conteneva, al di sotto della descrizione, anche una lista con le immagini dei personaggi con sfondo bianco e posa neutra

Ma per costruire uno scraper è necessario innanzitutto trovare un pattern secondo cui le



Figura 2.2: Esempio di Personaggi

Visual Novel sono organizzate e inoltre capire in che punto delle pagine si collocassero Siccome tutto questo potrebbe sembrare un progetto a sé stante abbiamo deciso di cercare in rete se ci fosse già uno scraper del sito...e fortunatamente lo abbiamo [trovato!](#)  
Ovviamente le cose non vanno mai come previsto e infatti questo scraper presentava dei problemi

### 2.3 Rifattorizzazione dello Scraper

Al momento di testare lo scraper ci siamo accorti che iniziava a fare il throw di alcune eccezioni riguardanti il fatto di non riuscire a scaricare le immagini

Inizialmente il codice non mostrava errori a runtime dato che era patchato con un sacco di try...except e quindi per capire dove fosse esattamente l'errore abbiamo deciso di rimuovere tutti questi costrutti ma non dal codice originale: abbiamo infatti deciso di riscrivere tutto lo scraper anche per capire se poteva presentare alcuni errori

Riscrivendo ci siamo accorti di alcune cose interessanti:

1. Alcune parti erano spesso ridondanti e quindi le abbiamo racchiuse in apposite funzioni
2. I nomi delle variabili erano criptici e non si riusciva a capire cosa rappresentassero
3. Lo scraper andava ad interrogare innanzitutto la pagina relativa ai giochi di un anno, poi interrogava le pagine dei mesi relativi a quell'anno e per ogni mese interrogava le pagine dei videogiochi. In queste ultime pagine erano presenti le immagini dei personaggi in tag img il cui attributo "src" faceva riferimento all'url di un server
4. Proprio il tentativo di accedere a questo url causava il crash dello scraper che non riusciva quindi a scaricare nessuna immagine

A cosa poteva essere dovuto tale errore?

Per capirlo abbiamo fatto delle prove manuali di download delle immagini e tutto filava liscio e quindi l'ipotesi da noi formulata è stata: il server per darci l'immagine vuole che la richiesta provenga dalla pagina del videogioco di cui l'immagine fa parte

A questo punto si è trattato solo di cambiare un parametro nel metodo che scaricava la pagina web mettendo appunto come provenienza della richiesta la pagina web del videogioco e finalmente lo scraper è riuscito a leggere la pagina con l'immagine

L'ultimo passaggio è stato estrarre il tag img da questa pagina e salvare l'immagine su Google Drive in modo da non ripetere sempre lo stesso procedimento

In figura 2.3 viene riportato un estratto dello scraper

```

print(f"Scraping images for year {year} and month {month}")
# Loop through the game types (pc or dvd)
for game_type in game_types:
    success = False
    retries = 0
    while not success:
        # try:
        #     Get all html tags of each game published that month of that year of that game_type
        game_elems = get_games_by_year_month(payload, year, month, game_type)
        # success = True
        # Loop through all game elements and get a game's url
        for game_elem in tqdm(game_elems):
            game_link = game_elem.find("a").attrs['href']
            # print(game_link)
            complete_game_url = root_url + game_link
            # print(complete_game_url)
            success = False
            retries = 0
            while not success:
                # try:
                #     Get the characters' tags of a single game
                game_page, character_tags = get_characters_tags(complete_game_url)
                # Loop through all the characters and retrieve their image
                for character in character_tags:
                    # print("Referer: " + game_page.url)
                    # print(character)
                    try:
                        img_bytestr = retrieve_image_from_url(character, referer_url=game_page.url)
                    except:
                        pass

                    # save_retrieved_image(dest_dir=all_images_month_dir, year=year, month=month, images_count=images_count)
                    save_retrieved_image(dest_dir=female_images_month_dir, year=year, month=month, images_count=images_count)

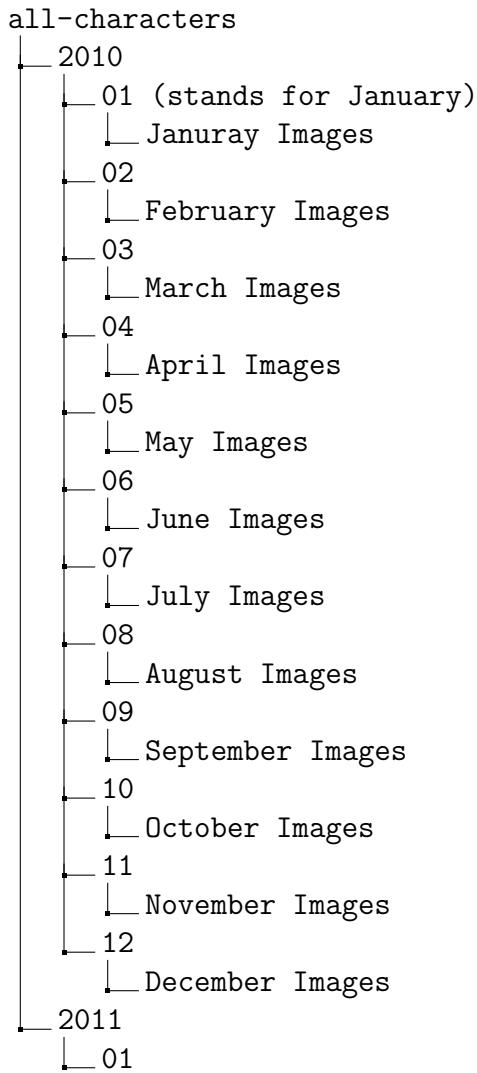
                    images_count += 1
                    success = True

```

Figura 2.3: Parte finale dello scraper

## 2.4 Organizzazione Dei File

Per seguire anche i cicli dello scraper abbiamo scelto di organizzare i file secondo la seguente struttura



E così via fino al 2021. Il paper in realtà aveva raccolto anche le immagini dal 2000, tuttavia, siccome le nostre risorse computazionali sono state notevolmente limitate, abbiamo deciso di saltare 10 anni di immagini, rinunciando quindi ad una migliore qualità di generazione

Stabilita quindi la struttura dei file abbiamo dovuto affrontare altri 2 problemi:

1. Estrarre solo le facce dei personaggi

2. Eliminare le immagini che non fossero ragazze, in modo da non avere rumore nel dataset

## 2.5 Rimuovere Le Non Ragazze

Prima di passare alla face detection abbiamo deciso di rimuovere prima dal dataset tutte quelle immagini che non contenessero ragazze in modo da avere rendere più leggero il secondo passaggio

Il problema principale è stato il seguente: come fa un algoritmo a capire se l'immagine che sta manipolando è una ragazza oppure no?

Per fare questo abbiamo utilizzato l'algoritmo [Illustration2Vec](#) che è una rete neurale pre-addestrata a riconoscere i tag di un'immagine assegnandoglieli con delle probabilità

Il suo utilizzo è stato molto semplice e intuitivo e quindi per discriminare le ragazze dalle non ragazze abbiamo adottato la seguente logica, una volta estratto il tag da ciascuna immagine del dataset:

1. Se l'immagine non ha il tag "1girl" allora questa viene immediatamente scartata
2. Se l'immagine possiede il tag "1girl" allora si guarda con che probabilità gli è stato assegnato e se è inferiore all'85% allora viene scartata
3. Se nessuna delle due condizioni si verifica l'immagine è una ragazza e quindi viene tenuta

In figura 2.4 viene mostrato il codice che implementa tale logica

```

illust2vec = i2v.make_i2v_with_chainer(
    "/content/illustration2vec/illust2vec_tag_ver200.caffemodel", "/content/illustration2vec/tag_list.json")

for year_dir in year_dirs:
    if year_dir not in scanned_years:
        year_path = super_path + "/" + year_dir
        try:
            os.mkdir(out_path + "/" + year_dir)
        except:
            pass
    month_dirs = os.listdir(year_path)

    for month_dir in month_dirs:
        print(f"Scanning Images of year {year_dir} and month {month_dir}")
        imgs_path = year_path + "/" + month_dir

        images = os.listdir(imgs_path)
        for img_path in tqdm(images):
            image_path = imgs_path + "/" + img_path
            img = Image.open(image_path)
            image_tags = illust2vec.estimate_plausible_tags([img], threshold=0.25)
            general_tags = image_tags[0]['general']

            is_a_girl = False
            for tag in general_tags:
                if tag[0] == 'igirl' and tag[1] >= 0.85:
                    is_a_girl = True
                    break

            if is_a_girl == False:
                #salva l'immagine nella cartella other_images
                img.save(out_path + "/" + year_dir + "/" + img_path)
                #elimina l'immagine attuale
                os.remove(image_path)

```

Figura 2.4: Estrazione Ragazze-NonRagazze

## 2.6 Face Detection e Extraction

Per rilevare ed estrarre i volti dalle immagini delle ragazze abbiamo utilizzato un cascade classifier, un algoritmo di object detection basato su ADA

Addestrare un classificatore di questo tipo è un compito abbastanza complesso, tuttavia sono spesso disponibili dei file XML con tutti i parametri necessari all'object detection per il task di interesse. Nel nostro caso abbiamo quindi trovato l'XML con i parametri necessari a trovare i volti dei personaggi anime

L'output di questo algoritmo conteneva una serie di valori tra cui le coordinate x e y del vertice in alto a sinistra del rettangolo e la sua larghezza e altezza

Di conseguenza il rettangolo con cui individuare il volto aveva le seguenti coordinate

1. x, y
2. x + width, y
3. x, y + height
4. x + width, y + height

Tuttavia ciò non bastava a individuare in modo soddisfacente i volti che non erano perfettamente centrati e a volti non erano compresi per intero

Nel paper di riferimento abbiamo infatti trovato che l'autore aveva ingrandito il rettangolo di 1.5 volte rispetto all'originale come mostrato in figura 2.5

I risultati ottenuti da questi due rettangoli sono mostrati in figura 2.6

Finalmente l'ultima cosa da fare era solo quella di costruire una struttura di cartelle del tipo

```
all-faces
  └── 2010
    └── 2010 images
  └── 2011
    └── 2011 images
```

E così via.

Fatto ciò siamo pronti per provare le nostre GAN ma, prima di farlo, vediamo come queste sono fatte e come operano

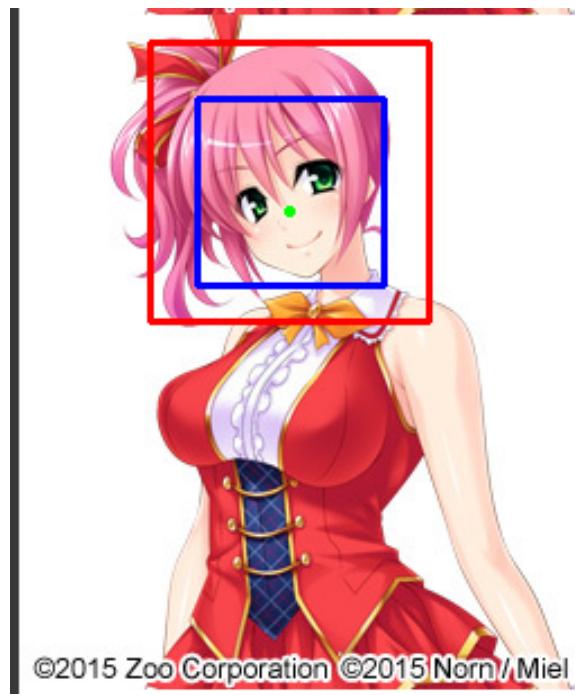


Figura 2.5: Esempio di rettangolo originale (blu) vs ingrandito di 1.5 volte (rosso)

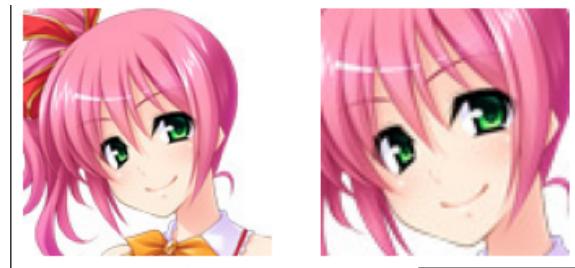


Figura 2.6: Sulla destra il volto estratto con il rettangolo rosso, sulla sinistra quello con il rettangolo blu

# 3. GAN

## 3.1 Cosa Sono le GAN

Semplicemente sono dei modelli generativi di Deep Learning

Ma che cos'è un modello generativo?

Questo è un modello che implicitamente o esplicitamente modella la distribuzione degli input e degli output siccome campionando da questi è possibile generare dati sintetici nell'input space

Un buon modello generativo può generare nuovi esempi che sono quasi indistinguibili da quelli veri

Quindi le GAN sono un'architettura per addestrare un modello generativo utilizzando il Deep Learning

Tale architettura ha 2 sotto modelli

1. Generator: modello usato per generare nuovi esempi dal dominio del problema
2. Discriminator: modello usato per classificare esempi come veri o falsi

Vediamo più nel dettagli come questi operano

### 3.1.1 Generator

Il generator riceve in ingresso un vettore di lunghezza fissa e genera un campione nel dominio, in questo caso quindi immagini. Il vettore in questione è preso casualmente da una distribuzione gaussiana ed è utilizzato come sorgente di rumore per il processo di generazione, per evitare che gli esempi generati siano sempre uguali e inoltre ha la particolarità di avere significato solo per il generatore

Lo spazio vettoriale da cui proviene è chiamato Latent Space e le variabili latenti sono semplicemente variabili casuali non osservabili direttamente

Spesso ci riferiamo alle variabili latenti come una proiezione o una compressione della distribuzione dei dati e quindi il Latent Space fornisce una compressione dei dati osservati.

Nel nostro caso il latent space si potrebbe vedere come un luogo in cui le immagini sono compresse e sono "unzippate" dal generator che è l'unico che sa come farlo  
Alla fine del suo training il generator viene ovviamente mantenuto

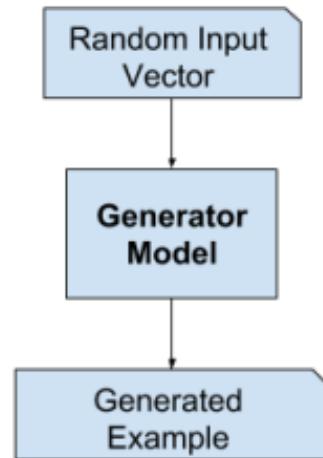


Figura 3.1: Modello del Generator

### 3.1.2 Discriminator

Il discriminator prende un esempio dal dominio del problema, un'immagine reale o generata, e fa una previsione in merito al fatto che questa sia vera (1) o falsa (0). Quella classificata come vera ovviamente viene dal dataset, quella falsa viene dal generator  
Alla fine del training il discriminator è scartato in quanto non ha più alcuna utilità

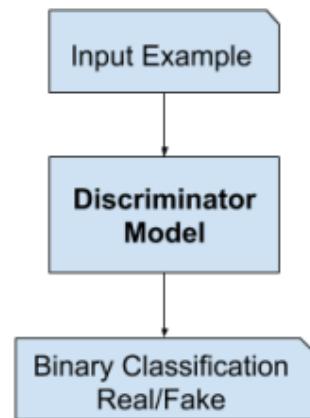


Figura 3.2: Modello del Discriminator

## 3.2 Le GAN: Un gioco a due giocatori

Nonostante il processo di un modello generativo sia unsupervised una proprietà alternativa di quest'architettura è che il training è modellato come un processo supervised: il generator e il discriminator sono addestrati contemporaneamente

Il generator crea un batch di campioni che sono dati in input al discriminator insieme ad esempi reali. Quest'ultimo poi provvede a classificarli come reali o fake ed è aggiornato per diventare più bravo in questo processo di discriminazione

Allo stesso modo il generator è aggiornato in base a quanto bene riesce ad eludere il discriminator facendogli sbagliare classificazione

Ecco che i due modelli sono "avversari" o "adversarial" nei termini della teoria dei giochi siccome stanno giocando un gioco a somma zero ossia quando il discriminator identifica correttamente esempi reali e fake è ricompensato e i suoi parametri non sono modificati mentre il generator è penalizzato con grossi update nei suoi parametri. Al contrario quando il generator riesce ad ingannare il discriminator è ricompensato senza cambiare i suoi parametri, mentre il discriminator è penalizzato aggiornando i suoi parametri

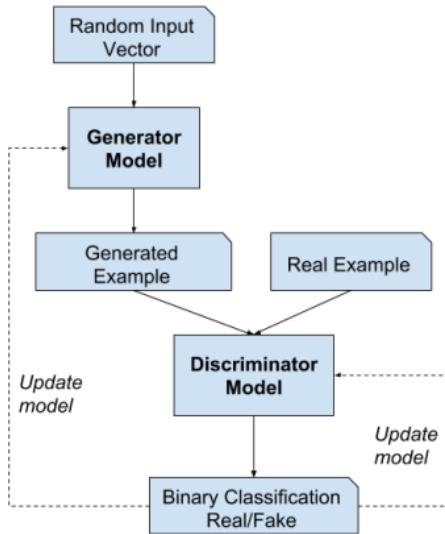


Figura 3.3: Gioco a due Giocatori a somma zero

### 3.3 Le GAN e le Reti Convoluzionali

Siccome le GAN lavorano tipicamente con immagini usano quindi le CNN (Convolutional Neural Networks) come Generator e Discriminator

Realizzare immagini significa che il Latent Space fornisce una rappresentazione compresa delle immagini di interesse

Ciò pone però un interrogativo, se non due:

1. Come fa il generatore a passare da una rappresentazione compressa ad un'immagine?
  2. Come funziona effettivamente il training di una GAN

Iniziamo subito a vedere il primo caso andando a parlare di Upsampling e Convoluzione Trasposta (o Deconvoluzione, termine però improprio come vedremo)

### 3.3.1 Upsampling

Supponiamo di avere una matrice  $2 \times 2$  e vogliamo trasformarla in una  $4 \times 4$ . Per rendere possibile ciò utilizziamo l'upsampling ripetendo gli elementi della matrice 2 volte ciascuno come mostrato in figura 3.4. Tutto qui, più facile a farsi che a dirsi.

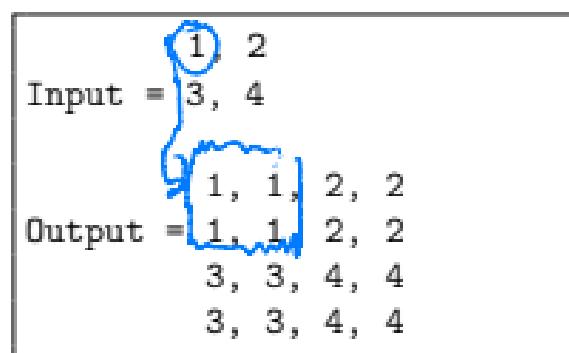


Figura 3.4: Esempio di Upsampling

Ora si potrebbe scegliere se implementare una propria versione oppure utilizzare i layer messi a disposizione da Pytorch o TensorFlow

La risposta sicura è: utilizzare le librerie siccome un problema di un approccio naïve è che i pixel dell'immagine che ha subito l'upsampling formeranno letteralmente dei blocchi di

colori. L'implementazione di una libreria andrebbe a ridurre questo problema utilizzando ad esempio tecniche di interpolazione bilineare per mitigare i colori di un blocco

Un vantaggio di utilizzare l'upsampling è che il generatore non avrebbe nessun parametro da imparare alleggerendo di molto i costi computazionali

Lo svantaggio è che, nonostante le tecniche di interpolazione, l'immagine risulta ancora molto a blocchi rispetto all'utilizzo di una convoluzione trasposta

### 3.3.2 Convoluzione Trasposta

Anche questo metodo esegue un upsampling ma invece di utilizzare poi una interpolazione tra i pixel per calcolare i colori va ad interpretare i dati di input attraverso una convoluzione  
Quindi Convoluzione Trasposta = Upsampling + Convoluzione

In questo metodo viene effettuato il mapping dalle features ai pixel invece che il contrario come accade con la convoluzione

Prima che però facciate brutta figura con un matematico è doveroso chiarire una cosa: la convoluzione a cui si fa riferimento in questa relazione non è quella vera e propria, il suo vero nome sarebbe Cross Correlation che è comunque un'operazione molto simile alla convoluzione e quindi per comodità la si chiama Convoluzione, ma volendo chiamare le cose con il loro vero nome non è così

Tornando al discorso sulla convoluzione trasposta: anche questa, come quella normale, chiede uno stride e ovviamente una kernel size oltre che un padding e tutto fa l'opposto della convoluzione tant'è che se volessimo potremmo anche fare una convoluzione con stride=1/2 per farne in realtà una trasposta con stride=2

Le figure 3.5 e 3.6 mostrano un esempio di convoluzione traposta con stride=2 Visti

---

$$\begin{bmatrix} [1 & 2] \\ [3 & 4] \end{bmatrix}$$

Figura 3.5: Matrice di Partenza

---


$$\begin{bmatrix} [1. 0. 2. 0.] \\ [0. 0. 0. 0.] \\ [3. 0. 4. 0.] \\ [0. 0. 0. 0.] \end{bmatrix}$$


---

Figura 3.6: Matrice dopo La Convoluzione Traposta con stride=2 e kernel=1

quindi i trucchi utilizzati dalle GAN, in particolare dal generator, prima di passare alle implementazioni, vediamo come funziona l'algoritmo di training di quest'architettura

### 3.4 Algoritmo Di Training Delle GAN

Iniziamo subito mostrando in figura 3.7 l'algoritmo presente sul paper di Goodfellow, il pioniere di questa architettura

---

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

---

```

for number of training iterations do
  for  $k$  steps do
    • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
    • Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
    • Update the discriminator by ascending its stochastic gradient:
      
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

  end for
  • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
  • Update the generator by descending its stochastic gradient:
    
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

```

---

Figura 3.7: Pseudo Codice Algoritmo Di Training

Possiamo notare come il ciclo più interno iteri lungo il range di iterazioni che però non è

un'epoca ma un'iterazione lungo gli elementi di un batch che porta all'update di generator e discriminator. Ricordiamo che un'epoca è invece una passata lungo tutto il dataset, suggerita dal for più esterno

Il processo di training consiste nel simulataneo SGD di generator e discriminator. In ogni step viene effettuato il campionamento di due minibatch: X dal dataset e z dalle variabili latenti

Per aggiornare il discriminator abbiamo i seguenti step:

1. Si creano i latent points da dare in input al generator che genera così dei fake
2. Si seleziona un batch dal dataset di training e lo si da in input al discriminator come "real"
3. Il discriminator utilizza poi gli esempi "fake" del generator per vedere se riesce a distinguerne la natura e aggiorna i suoi pesi come conseguenza

Infine va aggiornato il generator selezionando quindi un batch di latent points, generando dati "fake" che sono poi passati al discriminator e in base a quanto quest'ultimo riesce a indovinare i "fake" il generator subisce un aggiornamento dei pesi

### 3.4.1 GANs Loss Function

L'ultimo argomento trattato in questo lungo capitolo riguarda la funzione d'errore utilizzata per effettuare l'aggiornamento dei pesi

Il discriminator è addestrato per classificare immagini come "real" o "fake"

Ciò è reso possibile andando a massimizzare il logaritmo della probabilità di previsione delle immagini "real", insomma usano la cross-entropy

Tuttavia questa funzione non va bene per le GAN siccome da informazioni non buone sul gradiente che il generator può utilizzare per aggiustare i pesi e può condurlo a saturare

La funzione in questione è

$$\log(1 - D(G(z)))$$

che satura

Il generator si può quindi addestrare per massimizzare

$$\log(D(G(z)))$$

cercando quindi di massimizzare la probabilità che il discriminator sbagli mentre quest'ultimo cerca di massimizzare la sua probabilità di classificare correttamente dando vita ad un gioco denominato "minmax"

## 4. DCGAN

In questo capitolo verrà introdotta brevemente l'architettura della DCGAN [1] e in seguito verrà illustrato il processo di training utilizzato e infine i risultati che sono stati ottenuti. Le DCGAN o Deep Convolutional Generative Adversarial Network sono una classe di Reti neurali convoluzionali particolarmente adatte per l'unsupervised learning. Questa tecnica è stata proposta la prima volta nel 2016 e si basa sulle ricerche già svolte in ambito GAN ma inserisce tre modifiche fondamentali per rendere il training stabile su vari dataset con immagini a più alta risoluzione e permettendo di poter utilizzare un modello generativo più profondo.

La prima delle tre modifiche effettuate è stata quella di rimuovere i layer di pooling spaziale deterministico e rimpiazzarli con livelli di convoluzioni con stride maggiori di 1. Questo ha permesso ai modelli del generatore e del discriminatore di imparare il proprio upsampling e downsampling.

La seconda modifica è effettuata è stata quella di rimuovere i livelli fully connected finali della rete convoluzionale. Questi livelli che effettuavano una global average pooling aumentavano la stabilità ma riducevano il tempo di convergenza del modello. Inoltre il primo strato del generatore che prende come input una uniform noise distribution  $Z$ , potrebbe essere definito completamente connesso in quanto è solo una moltiplicazione di matrici. Questo risultato viene rimodellato in un tensore quadridimensionale e utilizzato come input allo stack convoluzionale. Per il discriminatore, l'ultimo strato di convoluzione viene appiattito e quindi inserito in un'unica uscita che utilizza la sigmoide come funzione di attivazione.

La terza modifica riguarda l'introduzione della Batch Normalization. Essa stabilizza l'apprentimento normalizzando l'input di ciascuna unità in modo che abbia media zero e varianza unitaria. Questo aiuta il training permettendo al gradiente di scorrere più facilmente lungo tutti i livelli profondi. La batch normalization tuttavia non va applicata al livello di output del generatore e a quello di input del discriminatore. Inoltre nel generatore viene utilizzata la funzione di attivazione ReLU ad eccezione del livello di output in cui

viene usata una tangente iperbolica. Nel discriminatore invece si fa uso della LeakyReLU per ciascun livello.

## 4.1 Implementazione

In questa sezione verrà presentato il codice utilizzato e verrà spiegato com'è stato effettuato il training. La architettura è stata implementata utilizzando la libreria PyTorch. Nella figura 4.1 viene mostrato il codice utilizzato per implementare il Discriminatore

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, bn=True):
            block = [nn.Conv2d(in_filters, out_filters, 3, 2, 1),
                     nn.LeakyReLU(0.2, inplace=True),
                     nn.Dropout2d(0.25)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block

        self.model = nn.Sequential(
            *discriminator_block(channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )

        ds_size = image_size // 2 ** 4
        self.adv_layer = nn.Sequential(
            nn.Linear(128 * ds_size ** 2, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        out = self.model(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)

        return validity
```

Figura 4.1: Discriminatore

Com'è possibile vedere il Discriminatore è composto da quattro livelli nascosti. Questi quattro livelli convoluzionali utilizzano una LeakyReLU come funzione di attivazione e un dropout dei pesi pari a 0.25. Inoltre nei 3 livelli successivi al primo, all'output del livello viene applicata una Batch Normalization. Successivamente a questi 4 livelli è presente un

livello fully connected che utilizza la sigmoide come funzione di attivazione.

Nella figura 4.2 è presente il codice del generatore

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = image_size // 4
        self.l1 = nn.Sequential(
            nn.Linear(latent_dim, 128 * self.init_size ** 2)
        )

        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )

    def forward(self, z):
        out = self.l1(z)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img = self.conv_blocks(out)
        return img
```

Figura 4.2: Generatore

Com'è possibile notare questa architettura è differente dalla precedente. Il primo livello effettua una trasformazione lineare: l'input con dimensione pari allo spazio latente viene trasformato nella dimensione ... In seguito il modello è composto da una serie di livelli di Batch Normalization, con dei livelli convoluzionali, dei livelli che implementano la LeakyReLU e infine un livello che implementa la tangente iperbolica

Nella figura 4.3 è riportato il codice utilizzato per effettuare il training della DCGAN. Inizialmente vengono inizializzati il generatore, il discriminatore e la loss. La funzione di Loss utilizzata nella DCGAN è la Binary Cross Entropy. Successivamente lo script verifica la presenza di una GPU. Se essa è disponibile allora l'esecuzione del generatore, del discriminatore e della loss vengono spostate sulla CPU. In seguito vengono inizializzati gli ottimizzatori utilizzati dal generatore e dal discriminatore. Entrambi sono ottimizzatori Adam.

I parametri utilizzati per Adam sono i seguenti:

- learning rate: 0.0002
- beta1: 0.5
- beta2: 0.999

```
save = True
adversarial_loss = torch.nn.BCELoss()
generator = Generator()
discriminator = Discriminator()

if cuda:
    generator.cuda()
    discriminator.cuda()
    adversarial_loss.cuda()

optimizer_G = torch.optim.Adam(generator.parameters(),
                               lr=lr, betas=(beta_1, beta_2))
optimizer_D = torch.optim.Adam(discriminator.parameters(),
                               lr=lr, betas=(beta_1, beta_2))

if (not (os.path.exists(models_path + "/DCGAN_model775.tar"))):
    print("training of a new model")
    generator.apply(weights_init_normal)
    discriminator.apply(weights_init_normal)

    train_GAN(n_epochs=50, index=0)
    generator.eval()

else:
    print("resuming training from a loaded model")
    checkpoint = torch.load(models_path + "/DCGAN_model775.tar")
    generator.load_state_dict(checkpoint['generator_state_dict'])
    discriminator.load_state_dict(checkpoint['discriminator_state_dict'])
    optimizer_G.load_state_dict(checkpoint['optimizer_G_state_dict'])
    optimizer_D.load_state_dict(checkpoint['optimizer_D_state_dict'])

    train_GAN(n_epochs=25, index=30) #change index everytime

if(save):
    save_GAN()
```

Figura 4.3: Script per effettuare un nuovo training oppure ripartire da un modello già allenato

Dopodichè lo script controlla se c'è un modello preallenato da caricare. Nel caso non ci sia, esso applica al generatore e al discriminatore una inizializzazione dei pesi che possiamo vedere nella figura 4.4

```
def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

Figura 4.4: Inizializzazione dei pesi

Successivamente all'inizializzazione dei pesi inizia il training di una nuova DCGAN. In alternativa, qualora il modello sia stato precedentemente allenato, lo script provvederà a caricare il checkpoint del training precedente e incomincerà una nuova sessione di training.

Per effettuare il training questo script chiama il metodo trainGAN che possiamo vedere nella figura 4.5

```

def train_GAN(n_epochs = 200, index = 0):
    d_hist, g_hist = list(), list()

    for epoch in range(n_epochs):

        i = 0
        for imgs, _ in tqdm(dataloader):

            valid = Variable(Tensor(imgs.shape[0], 1).fill_(1.0), requires_grad=False)
            fake = Variable(Tensor(imgs.shape[0], 1).fill_(0.0), requires_grad=False)

            real_imgs = Variable(imgs.type(Tensor))

            optimizer_G.zero_grad()

            z = Variable(Tensor(np.random.normal(0,1,(imgs.shape[0], latent_dim)))) 

            gen_imgs = generator(z)

            g_loss = adversarial_loss(discriminator(gen_imgs), valid)

            g_loss.backward()
            optimizer_G.step()

            optimizer_D.zero_grad()

            real_loss = adversarial_loss(discriminator(real_imgs), valid)
            fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
            d_loss = (real_loss + fake_loss) / 2

            d_loss.backward()
            optimizer_D.step()

            d_hist.append(d_loss.item())
            g_hist.append(g_loss.item())

            i += 1

            batches_done = epoch * len(dataloader) + i
            if batches_done % (len(dataloader)*1) == 0:
                save_image(gen_imgs.data[:16], f"{results_path}/{index}_{batches_done}.png", nrow=4, normalize=True)
                print(
                    "[Epoch %d %d] [Batch %d %d] [D loss: %f] [G loss: %f]"
                    %(epoch, n_epochs, i, len(dataloader), d_loss.item(), g_loss.item())
                )
                plot_history(d_hist, g_hist, index, models_path)

```

Figura 4.5: Fase di training

Il training della DCGAN è stato svolto in un intervallo temporale di circa 20 giorni. Per questioni di tempo e di permessi di colab, è stato scelto di addestrare la DCGAN per circa 50 epoche ogni giorno per un totale di 800 epoche. L’addestramento del modello ha richiesto giornalmente all’incirca dalle 2 alle 3 ore di tempo. Il salvataggio della DCGAN per poter ricominciare il training in un secondo momento è stato di vitale importanza, dato che Google Colab dopo ore di utilizzo termina automaticamente il runtime non permettendoci di arrivare alle 200 epoche che sono state prefissate come minime per poter ottenere risultati accettabili. Come è possibile vedere dal grafico, alla fine di ogni epoca la DC-GAN stampa il grafico della Loss e un’immagine contenente le figure da essa generate. I grafici e le immagini verranno mostrate nella prossima sezione. Oltre alla generazione

delle immagini, si è scelto di esplorare lo spazio latente generando una GIF.

## 4.2 Risultati

L'ultima parte di questo capitolo è dedicata a mostrare alcuni risultati ottenuti dopo 800 epoche di training di questo modello Verrano mostrati in particolare:

1. Risultato dopo 1 epoca
2. Risultato dopo 10 epoche
3. Risultato dopo 100 epoche
4. Risultato dopo 200 epoche
5. Risultato dopo 400 epoche
6. Risultato dopo circa 800 epoche
7. Comportamento delle Loss Functions di generator e discriminator

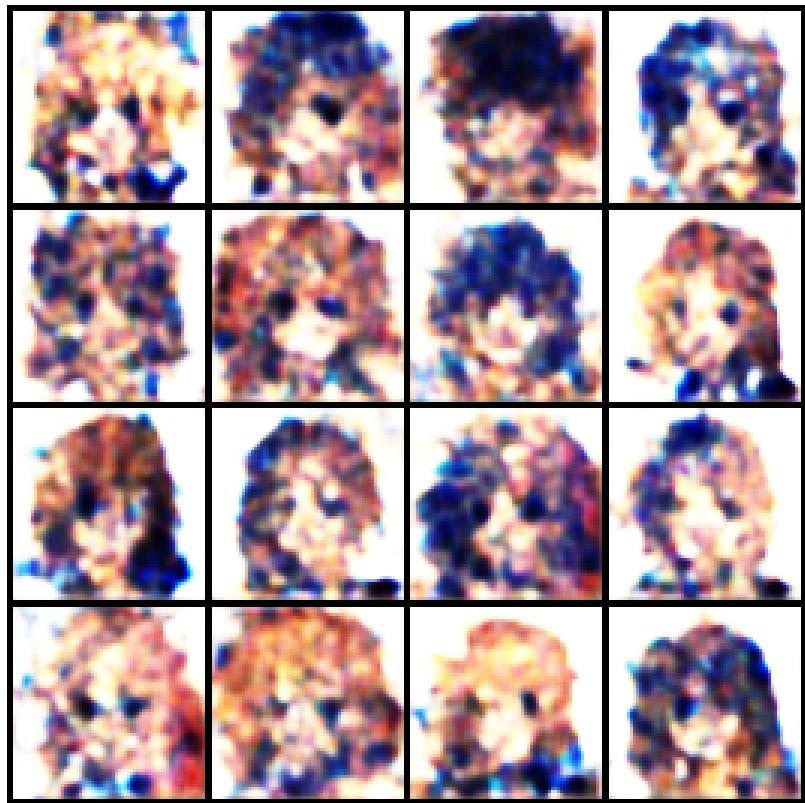


Figura 4.6: Risultato dopo 1 epoca

Nella figura 4.6 è possibile vedere i risultati prodotti dalla DCGAN dopo la prima epoca. Le figure rappresentate non ricordano ancora dei personaggi femminili ma sembrano più un insieme di pixel.



Figura 4.7: Risultato dopo 10 epoche

La situazione dopo 10 epoche migliora (Figura 4.7). In questa epoca è già possibile distinguere delle parvenze umanoidi poichè è possibile distinguere i capelli dalla faccia.



Figura 4.8: Risultato dopo 100 epoche

Dopo 100 epoche le figure femminili appaiono molto più nitide. La DCGAN ha imparato i colori e sta ora imparando le forme dei capelli e del viso.



Figura 4.9: Risultato dopo 200 epoche

Dopo 200 epoche la situazione migliora ulteriormente. I capelli risultano molto più nitidi e dettagliati, anche se con alcune imperfezioni in qualche immagine. Alcune immagini iniziano ad essere accettabili mentre altre presentano ancora irregolarità



Figura 4.10: Risultato dopo 400 epoch

Dopo 400 epoch possiamo notare come le figure femminili siano migliorate notevolmente. Il numero di immagini generate senza particolari artefatti è salito. I colori e i dettagli dei capelli sembrano ulteriormente migliorati e gli occhi posizionati nella maniera corretta nella maggior parte dei casi



Figura 4.11: Risultato dopo circa 800 epocha

Dopo 800 iterazioni è possibile notare le immagini siano peggiorate. Esse risultano molto più nitide ma anche più distorte

Qui sotto sono riportati i grafici che mostrano gli andamenti delle loss per le varie epocha

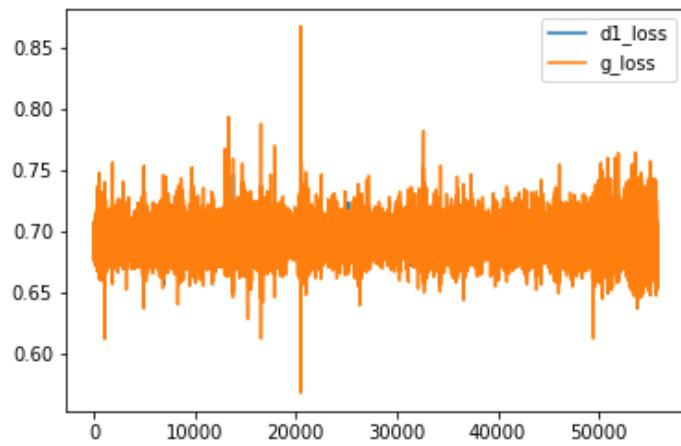


Figura 4.12: Grafico delle loss function per le epoche dalla 50 alla 100

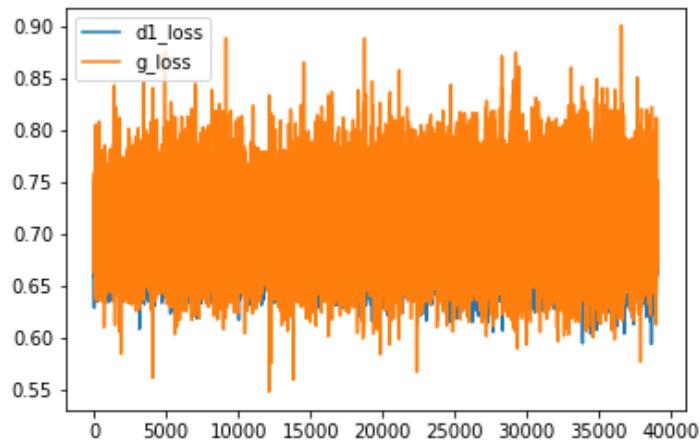


Figura 4.13: Grafico delle loss function per le epoche dalla 165 alla 200

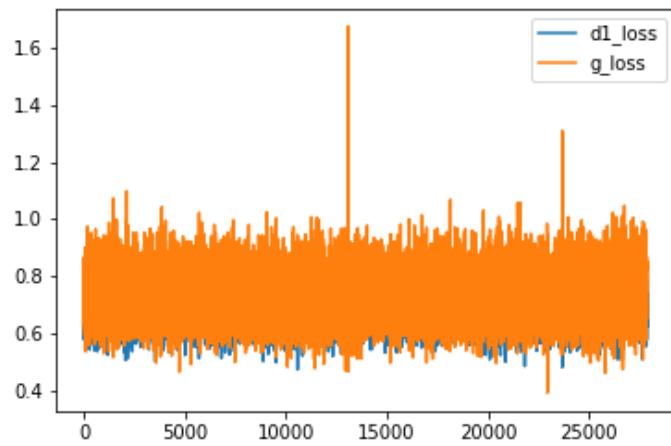


Figura 4.14: Grafico delle loss function per le epoche dalla 375 alla 400

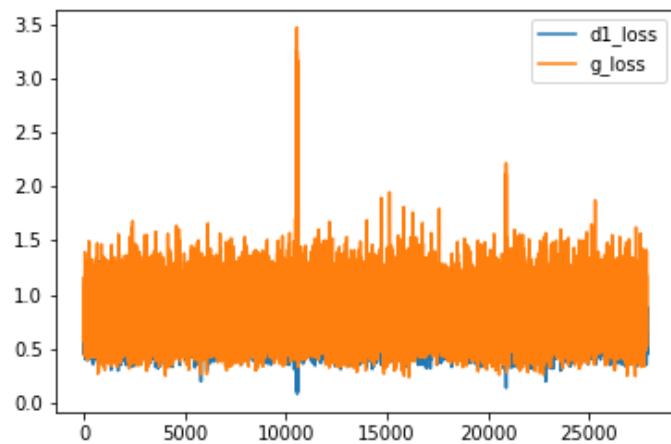


Figura 4.15: Grafico delle loss function per le epoche dalla 775 alla 800

## 5. WGAN

La WGAN è una architettura che presenta diverse differenze principali rispetto alla DC-GAN. Sono due gli aspetti che si sono voluti ottenere con questa nuova architettura:

- Avere una loss che correlasse correttamente il generatore con le immagini da esso prodotte
- Aumentare la stabilità del training

Per ottenere questo è stato necessario introdurre 3 cambiamenti dentro l'architettura classica.

1. Utilizzare una loss basata sulla distanza Earth-Mover
2. Addestrare il generatore solamente durante delle epoche critiche
3. Effettuare un clipping dei pesi del discriminatore

Nel nostro problema il generatore viene addestrato ogni epoca critica poichè il discriminatore riconosce facilmente una immagine reale da una falsa. Questo porterebbe ad un continuo aggiornamento dei pesi del generatore, causando il fenomeno di vanishing gradient. Riducendo il numero di iterazioni in cui il generatore viene addestrato il problema viene evitato.

---

**Algorithm 1** WGAN, our proposed algorithm. All experiments in the paper used the default values  $\alpha = 0.00005$ ,  $c = 0.01$ ,  $m = 64$ ,  $n_{\text{critic}} = 5$ .

---

**Require:** :  $\alpha$ , the learning rate.  $c$ , the clipping parameter.  $m$ , the batch size.  
 $n_{\text{critic}}$ , the number of iterations of the critic per generator iteration.  
**Require:** :  $w_0$ , initial critic parameters.  $\theta_0$ , initial generator's parameters.

```

1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSPProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSPProp}(\theta, g_\theta)$ 
12: end while
```

---

Figura 5.1: Algoritmo di training della WGAN

## 5.1 Implementazione

Qui sotto sono riportate le immagini relative all'implementazione utilizzando Pytorch. In figura 5.2 è possibile vedere l'implementazione del Generatore, mentre nella figura 5.3 è possibile vedere l'implementazione del discriminatore

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

    def block(in_feat, out_feat, normalize=True):
        layers = [nn.Linear(in_feat, out_feat)]
        if normalize:
            layers.append(nn.BatchNorm1d(out_feat, 0.8))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        return layers

    self.model = nn.Sequential(
        *block(latent_dim, 128, normalize=False),
        *block(128, 256),
        *block(256, 512),
        *block(512, 1024),
        nn.Linear(1024, int(np.prod(img_shape))),
        nn.Tanh()
    )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.shape[0], *img_shape)
        return img
```

Figura 5.2: Generator

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
        )

    def forward(self, img):
        img_flat = img.view(img.shape[0], -1)
        validity = self.model(img_flat)

        return validity

```

Figura 5.3: Discriminator

In questo capitolo non è verrà spiegato nuovamente lo script che chiama trainGAN su un nuovo modello oppure che ne carica uno precedentemente addestrato. Per esso si faccia riferimento alla figura 4.3. L'unica differenza in questo caso è quella che la WGAN non utilizza l'ottimizzatore Adam ma RMSprop con learning rate pari a 0.0002

Anche lo script in figura 5.4 è pressoché identico allo script utilizzato per il training della DCGAN.

Le differenze degne di nota sono 3:

- Il calcolo della Loss
- Il fatto che viene effettuato il weightclipping
- Il training del generator viene effettuato ogni 5 epoche

```

def train_GAN(n_epochs = 200, index = 0):
    batches_done = 0
    clip_value = 0.01
    n_critic = 5
    d_hist, g_hist = list(), list()
    for epoch in range(n_epochs):

        i = 0
        for imgs, _ in tqdm(dataloader):

            # Configure input
            real_imgs = Variable(imgs.type(Tensor))

            # -----
            # Train Discriminator
            # -----

            optimizer_D.zero_grad()

            # Sample noise as generator input
            z = Variable(Tensor(np.random.normal(0, 1, (imgs.shape[0], latent_dim))))

            # Generate a batch of images
            fake_imgs = generator(z).detach()
            # Adversarial loss
            loss_D = -torch.mean(discriminator(real_imgs)) + torch.mean(discriminator(fake_imgs))

            loss_D.backward()
            optimizer_D.step()

            # Clip weights of discriminator
            for p in discriminator.parameters():
                p.data.clamp_(-clip_value, clip_value)

            # Train the generator every n_critic iterations
            if i % n_critic == 0:
                # -----
                # Train Generator
                # -----

                optimizer_G.zero_grad()

                # Generate a batch of images
                gen_imgs = generator(z)
                # Adversarial loss
                loss_G = -torch.mean(discriminator(gen_imgs))

                loss_G.backward()
                optimizer_G.step()

                g_hist.append(loss_G.item())
                d_hist.append(loss_D.item())

            i += 1

            batches_done = epoch * len(dataloader) + i

            if batches_done % (len(dataloader)) == 0:
                save_image(gen_imgs.data[:25], f"{results_path}/{index}_{batches_done}.png", nrow=5, normalize=True)
                print(
                    f"[Epoch {epoch}/{n_epochs}] [Batch {batches_done}/{len(dataloader)}] [D loss: {loss_D.item():.4f}] [{G loss: {loss_G.item():.4f}}]"
                    f"\n{epoch}, {n_epochs}, {batches_done % len(dataloader)}, {len(dataloader)}, {loss_D.item()}, {loss_G.item()}"
                )
                plot_history(d_hist, g_hist, index, models_path)

```

Figura 5.4: Training

Il training, come per la DCGAN, è stato effettuato eseguendo 50 epochhe ogni giorno. Le tempistiche di esecuzione dell'algoritmo sono simili e in totale sono stati richiesti circa 4 giorni per arrivare alle 200 epochhe. Anche qui si è deciso di esplorare lo spazio latente, generando una GIF

## 5.2 Risultati

Qui sotto sono riportati i risultati ottenuti dopo 200 epochhe di training di questo modello. Verrano mostrati in particolare:

1. Risultato dopo 10 epochhe
2. Risultato dopo 100 epochhe
3. Risultato dopo 200 epochhe
4. Comportamento delle Loss Functions di generator e discriminator



Figura 5.5: Risultato dopo 10 epochhe

I risultati ottenuti dopo 10 epochhe mostrano già come sia differente la generazione delle immagini. Queste risultano molto meno nitide e sembra quasi siano state disegnate utilizzando i pastelli.

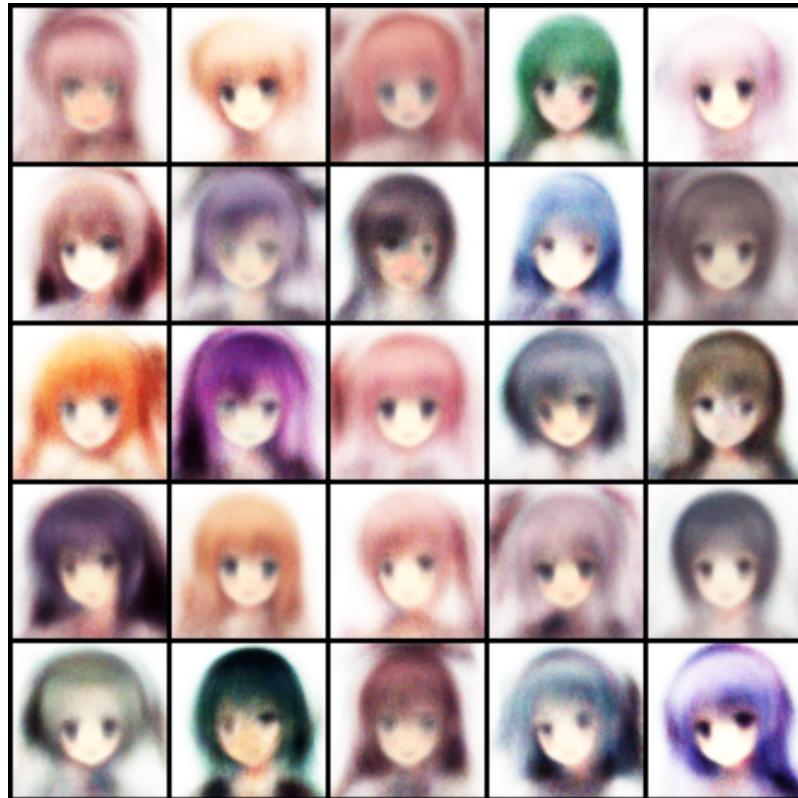


Figura 5.6: Risultato dopo 100 epochhe

Come possiamo vedere il risultato dopo 100 epochhe è molto simile a quello ottenuto dopo 10 epochhe



Figura 5.7: Risultato dopo 200 epochhe

Il risultato dopo 200 epochhe non migliora rispetto ai risultati visti precedentemente. Da questo possiamo concludere che questa architettura non è particolarmente adatta per il problema posto. Non avendo visto risultati promettenti, abbiamo deciso di non approfondire ulteriormente il training

Nelle immagini sottostanti è possibile vedere l'andamento della loss nelle prime 50 e nelle ultime 50 epochhe. Come è possibile vedere il valore delle loss oscilla tra valori molto alti a valori molto bassi

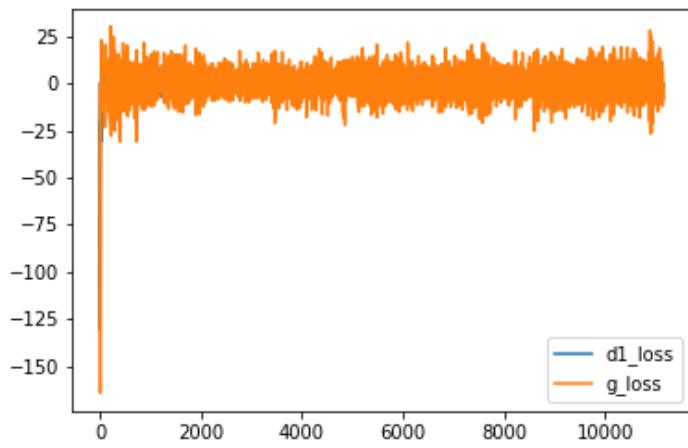


Figura 5.8: Grafico delle loss function per le epoche dalla 0 alla 50

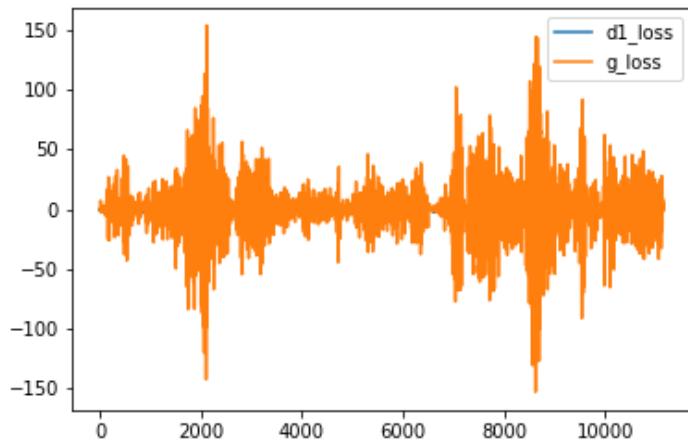


Figura 5.9: Grafico delle loss function per le epoche dalla 150 alla 200

## 6. WGAN-GP

Come visto nello scorso capitolo la WGAN (Wasserstein GAN) utilizza la distanza di Wasserstein come loss function per produrre risultati migliori dell'originale GAN con il discriminator che deve produrre una funzione 1-Lipshitz forzata però attraverso il clipping dei pesi a dei valori

Tale GAN (paper [2]) apporta dei miglioramenti alla WGAN in questo senso:

1. Mostra come il clipping dei pesi possa portare a dei risultati indesiderati
2. Propone la Penalità del Gradiente (da cui il nome WGAN-GP) che non soffre di questi problemi
3. Permette un training più stabile della GAN

### 6.1 Wasserstein GAN

Come detto nel capitolo precedente la WGAN si appoggia su un'altra loss function chiamata Earth Mover o Wasserstein Loss mostrata in figura 6.1

$$\min_G \max_{D \in \mathcal{D}} \mathbb{E}_{\mathbf{x} \sim \mathbb{P}_r} [D(\mathbf{x})] - \mathbb{E}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g} [D(\tilde{\mathbf{x}})]$$

Figura 6.1: Earth Move Distance o Wasserstein Loss

In questo caso  $D$  è l'insieme delle funzioni 1-Lipschitz e  $P_g$  è invece la distribuzione del modello definita come  $x = G(z), z = p(z)$ . Tale funzione produce un gradiente rispetto all'input si comporta meglio rispetto a quello della GAN classica rendendo l'ottimizzazione del generator più semplice

Tuttavia questo metodo per mantenere il discriminator una funzione 1-Lipshitz propone di utilizzare il clipping dei pesi in un range  $[-c, c]$  che porta a difficoltà di ottimizzazione. In alcuni casi questi problemi si possono evitare utilizzando la Batch Normalization nel discriminator, tuttavia con GAN che presentano molti layers anche utilizzare questo tipo di

approccio porta ad una divergenza del training

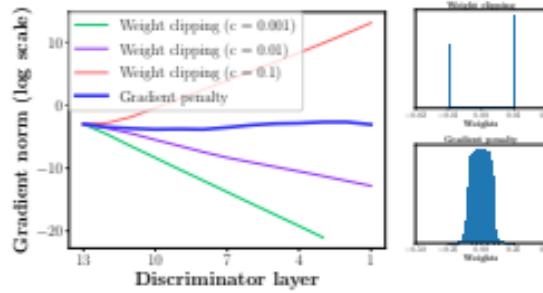


Figura 6.2: Divergenze con Weight Clipping Rispetto a Gradient Penalty

Quindi implementare una qualsiasi funzione k-Lipshitz attraverso il weight clipping rende il discriminator tendente ad imparare funzioni molto semplici perdendosi i momenti nella distribuzioni dei dati ma limitandosi a modellare semplici approssimazioni della funzione ottimale

Un altro problema del weight clipping è che senza un opportuno tuning del parametro  $c$  il training può comparire il problema del Vanishing/Exploding Gradient come mostrato in figura 6.2

## 6.2 Utilizzo della Penalità Del Gradiente

Per rendere il discriminator una funzione 1-Lipshitz, ossia una funzione il cui gradiente ha norma 1 ovunque, si è pensato di vincolare la norma del gradiente del discriminator rispetto al suo input. Tale risultato è una nuova loss function che introduce un termine di regolarizzazione alla loss originale del discriminator della WGAN come mostrato in figura 6.3

Di seguito è infine riportato l'algoritmo della WGAN-GP molto simile a quello della WGAN ma con l'introduzione della regolarizzazione. In figura 6.4 viene infine mostrato l'algoritmo di training della WGAN-GP

$$L = \underbrace{\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} [D(\tilde{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Original critic loss}} + \lambda \underbrace{\mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Our gradient penalty}}.$$

Figura 6.3: WGAN-GP Loss

---

**Algorithm 1** WGAN with gradient penalty. We use default values of  $\lambda = 10$ ,  $n_{\text{critic}} = 5$ ,  $\alpha = 0.0001$ ,  $\beta_1 = 0$ ,  $\beta_2 = 0.9$ .

---

**Require:** The gradient penalty coefficient  $\lambda$ , the number of critic iterations per generator iteration  $n_{\text{critic}}$ , the batch size  $m$ , Adam hyperparameters  $\alpha, \beta_1, \beta_2$ .  
**Require:** initial critic parameters  $w_0$ , initial generator parameters  $\theta_0$ .

```

1: while  $\theta$  has not converged do
2:   for  $t = 1, \dots, n_{\text{critic}}$  do
3:     for  $i = 1, \dots, m$  do
4:       Sample real data  $x \sim \mathbb{P}_r$ , latent variable  $z \sim p(z)$ , a random number  $\epsilon \sim U[0, 1]$ .
5:        $\tilde{x} \leftarrow G_\theta(z)$ 
6:        $\hat{x} \leftarrow \epsilon x + (1 - \epsilon)\tilde{x}$ 
7:        $L^{(i)} \leftarrow D_w(\hat{x}) - D_w(x) + \lambda(\|\nabla_{\hat{x}} D_w(\hat{x})\|_2 - 1)^2$ 
8:     end for
9:      $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$ 
10:   end for
11:   Sample a batch of latent variables  $\{z^{(i)}\}_{i=1}^m \sim p(z)$ .
12:    $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^m -D_w(G_\theta(z)), \theta, \alpha, \beta_1, \beta_2)$ 
13: end while
```

---

Figura 6.4: Algoritmo di Training della WGAN-GP

## 6.3 Implementazione

Discussa quindi la natura teorica di questa GAN in questa sezione vengono riportati i punti del codice che implementano il generator e il discriminator insieme all'algoritmo di training e della Gradient Penalty

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

    def block(self, in_feat, out_feat, normalize=True):
        layers = [nn.Linear(in_feat, out_feat)]
        if normalize:
            layers.append(nn.BatchNorm1d(out_feat, 0.8))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        return layers

    self.model = nn.Sequential(
        *block(latent_dim, 128, normalize=False),
        *block(128, 256),
        *block(256, 512),
        *block(512, 1024),
        nn.Linear(1024, int(np.prod(img_shape))),
        nn.Tanh()
    )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.shape[0], *img_shape)
        return img
```

Figura 6.5: Generator

Come si può vedere nella figure 6.5, 6.6, 6.7 i modelli e l'algoritmo di training non sono niente di nuovo rispetto a quelli già mostrati, mentre la figura 6.8 mostra l'algoritmo di gradient penalty introdotto dalla WGAN-GP

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
        )

    def forward(self, img):
        img_flat = img.view(img.shape[0], -1)
        validity = self.model(img_flat)
        return validity

```

Figura 6.6: Discriminator

```

z = Variable(Tensor(np.random.normal(0,1,(imgs.shape[0], latent_dim)))) 

fake_imgs = generator(z)

real_validity = discriminator(real_imgs)

fake_validity = discriminator(fake_imgs)

gradient_penalty = compute_gradient_penalty(discriminator, real_imgs.data, fake_imgs.data)

d_loss = -torch.mean(real_validity) + torch.mean(fake_validity) + lambda_gp * gradient_penalty

d_loss.backward()
optimizer_D.step()

optimizer_G.zero_grad()

if i % n_critic == 0:

    fake_imgs = generator(z)

    fake_validity = discriminator(fake_imgs)
    g_loss = -torch.mean(fake_validity)

    g_loss.backward()
    optimizer_G.step()

    g_hist.append(g_loss.item())
    d_hist.append(d_loss.item())

```

Figura 6.7: Training

```

def compute_gradient_penalty(D, real_samples, fake_samples):
    """Calculates the gradient penalty loss for WGAN GP"""
    # Random weight term for interpolation between real and fake samples
    alpha = Tensor(np.random.random((real_samples.size(0), 1, 1, 1)))
    # Get random interpolation between real and fake samples
    interpolates = (alpha * real_samples + ((1 - alpha) * fake_samples)).requires_grad_(True)
    d_interpolates = D(interpolates)
    fake = Variable(Tensor(real_samples.shape[0], 1).fill_(1.0), requires_grad=False)
    # Get gradient w.r.t. interpolates
    gradients = autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True
    )[0]
    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty

```

Figura 6.8: Algoritmo di Gradient Penalty

## 6.4 Risultati

L'ultima parte di questo capitolo è ora dedicata a mostrare alcuni risultati ottenuti dopo 400 epoche di training di questo modello

Verrano mostrati in particolare:

1. Risultato dopo circa 10 epoche
2. Risultato dopo circa 100 epoche
3. Risultato dopo 200 epoche
4. Risultato dopo 400 epoche
5. Comportamento delle Loss Functions di generator e discriminator



Figura 6.9: Risultati dopo 10 epoche

Come mostrato nelle figure 6.9, 6.10, 6.11, 6.12 c'è un progressivo miglioramento nei risultati fino alle 200 epoche circa. Da qui in poi i miglioramenti sono molto minimi e ad



Figura 6.10: Risultati dopo 100 epoche



Figura 6.11: Risultati dopo 200 epoche



Figura 6.12: Risutati dopo 400 epoche

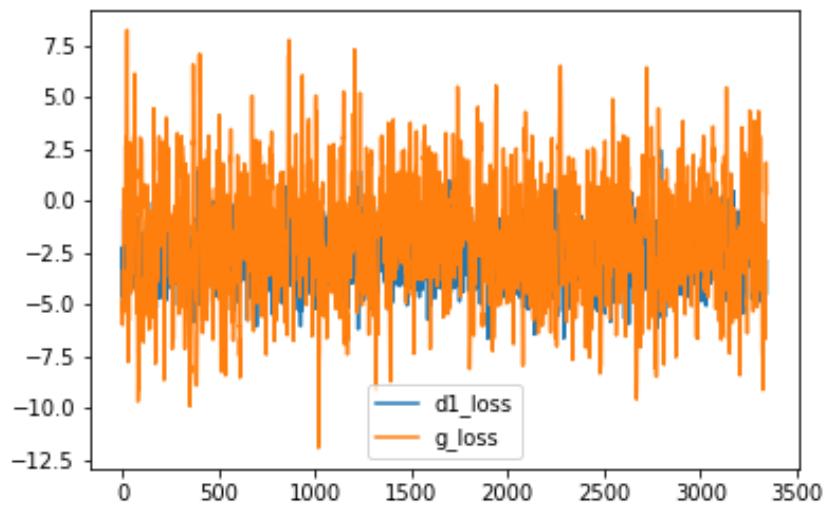


Figura 6.13: Andamento generale della loss

andare a 400 infatti i risultati non sono migliorati

La figura 6.13 riporta invece l'andamento della loss function che notiamo oscillare pa-

recchio lato generator e un po' meno lato discriminator (al punto che la prima copre la seconda). Questo andamento è perfettamente normale e indica una stabilità nel training della GAN

## 7. Conclusioni e sviluppi futuri

In conclusione possiamo dire di aver raggiunto gli obiettivi prefissati. Siamo riusciti ad emulare il paper, ottenendo risultati che seppur di qualità inferiore, risultano comunque soddisfacenti. Siamo riusciti a vedere in azione le GAN, a capire le differenze tra le loro architetture, come funziona il loro training e in che modo le immagini venissero generate. Inoltre questo progetto ci ha permesso di capire come generare il dataset delle immagini e di che dimensione dovesse essere per poter ottenere risultati soddisfacenti.

Per sviluppare ulteriormente questo progetto si potrebbe provare ad addestrare una Style-Gan e analizzare i risultati ottenuti. Un altro aspetto che è degno di essere approfondito è quello delle Conditional GAN per poter decidere le caratteristiche fisiche che i personaggi generati devono avere. In ultima istanza, si potrebbe creare un sito web in cui inserire questi modelli di GAN pronti per essere utilizzati dai fumettisti per generare immagini e trarre ispirazione da esse. Ringraziamo il Chiarissimo Professore Serina per la splendida opportunità che ci ha concesso

## Bibliografia

- [1] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.
- [2] Martin Arjovsky Vincent Dumoulin Aaron Courville1 Ishaan Gulrajani, Faruk Ahmed. Improved training of wasserstein gan. *arXiv preprint arXiv:1704.00028v3*, 2017.