**UNIVERSITÀ POLITECNICA DELLE MARCHE**

FACOLTÀ DI INGEGNERIA

---

Corso di Laurea triennale in *Ingegneria Informatica e dell'Automazione*

# Progettazione e sviluppo di un modello GAN per l'augmentation adattiva di dataset per la segmentazione.

*Design and development of a GAN model for adaptive augmentation of segmentation datasets.*

**Relatore:**
Prof. Adriano Mancini

**Laureando:**
Massimiliano Biancucci

---

Anno Accademico 2022/2023

# Prefazione

Il mio percorso nel campo dell'intelligenza artificiale è iniziato diversi anni fa, alle superiori per l'esattezza, dove sentii per la prima volta parlare di reti neurali, ad un corso pomeridiano voluto dal prof. Roberto Lulli il quale mi ha mostrato per primo questo affascinante campo di ricerca.

Ho svolto durante il mio percorso di studi diversi progetti incentrati su questa tematica, partendo da semplici reti neurali, e confrontandomi con progetti sempre più complessi fino ad arrivare ai modelli generativi basati sull'architettura GAN (Generative Adversarial Network), del quale in questa tesi proporrò una variante.

Lo scopo di questa tesi è quello di investigare la fattibilità di una potenziale soluzione ad uno dei grandi problemi che affligge oggi le aziende che si occupano di addestrare modelli neurali per la segmentazione di immagini. Tale problema è rappresentato dalle difficoltà e dai costi che vanno affrontati per creare dataset per specifici task, necessari per effettuare l'addestramento, e dunque la messa in produzione di tali modelli.

Tale scelta è stata naturale, in quanto ho dovuto confrontarmi in prima persona con questo problema nell'ultimo anno, come sviluppatore presso l'azienda Cloe.ai. In questa esperienza ho partecipato alla gestione, per quasi un anno, della realizzazione di un complesso dataset per l'addestramento di un modello di segmentazione, tale dataset aveva dei requisiti molto stringenti, e al contempo erano state assegnate al progetto una quantità limitata di risorse.

La realizzazione di un dataset su larga scala è un'operazione molto complessa, che richiede una elevata coordinazione tra annotatori, revisori, sviluppatori, e un'accurata documentazione che in base al problema può richiedere anche diversi mesi per poter essere redatta efficacemente. Tutto ciò mi ha fornito la motivazione per cercare una soluzione per accorciare questo lungo e tedioso processo e dunque attenuare gli ingenti costi che un'azienda deve sostenere per realizzare un dataset di questo tipo.

# Indice

**Prefazione** iii

**Indice** v

**1 Introduzione** 1

1.1 Motivazione . . . . . 1

1.1.1 Modelli neurali per la segmentazione nel controllo qualità . . . . . 1

1.1.2 Il dataset, requisiti e problematiche di realizzazione . . . . . 2

1.1.3 Approccio al problema . . . . . 3

1.2 La nascita del deep learning . . . . . 6

1.2.1 Dal machine learning al deep learning . . . . . 6

1.2.2 Le reti neurali biologiche . . . . . 7

1.3 Le reti neurali feed forward . . . . . 7

1.3.1 Il neurone artificiale . . . . . 8

1.3.2 Il Back-propagation . . . . . 10

1.3.3 Teorema di approssimazione universale . . . . . 12

1.4 Le reti neurali convoluzionali . . . . . 13

1.4.1 Storia delle CNN . . . . . 13

1.4.2 La convoluzione . . . . . 15

1.4.3 I parametri della convoluzione . . . . . 18

1.4.4 Il pooling . . . . . 20

1.4.5 La convoluzione multichannel . . . . . 22

**2 Stato dell'arte** 23

2.1 Il primo modello GAN . . . . . 23

2.1.1 L'adversarial training . . . . . 23

2.1.2 La loss function . . . . . 25

2.1.3 La convergenza del generatore . . . 27
2.1.4 Il gradient vanishing . . . 28
2.1.5 Il mode collapse . . . 28
2.1.6 Alcuni risultati . . . 29
2.2 DCGAN . . . 30
2.2.1 L'architettura del modello . . . 30
2.2.2 L'algebra vettoriale nello spazio latente Z . . . 31
2.3 Wasserstein GAN . . . 32
2.3.1 Earth-Mover distance . . . 33
2.3.2 Alcuni risultati . . . 35
2.4 Pix2Pix . . . 36
2.4.1 La loss function . . . 36
2.4.2 L'architettura . . . 37
2.4.3 Alcuni esempi di applicazione . . . 38
2.5 Large Mask inpainting with fast fourier convolution: LaMa . . . 39
2.5.1 Struttura del modello . . . 39
2.5.2 La loss function . . . 41
2.5.3 Alcuni risultati . . . 45

**3 Materiali e metodi** **47**
3.1 Il Dataset: Severstal steel defect detection . . . 47
3.1.1 Distribuzione del dataset . . . 48
3.2 Librerie e Framework . . . 52
3.2.1 Numpy . . . 52
3.2.2 OpenCV . . . 52
3.2.3 PyTorch . . . 52
3.2.4 Distributed data parallel . . . 52
3.3 Google cloud compute instance . . . 53
3.4 Repository del progetto . . . 53

**4 Sviluppo del progetto** **55**
4.1 Definizione della pipeline di addestramento . . . 55
4.2 Preparazione dei dati per la pipeline . . . 57
4.2.1 Severstal steel defect detection dataset reader . . . 57

4.2.2 Conversione nel formato line json . . . . . . 58
4.2.3 Split in train e test set . . . . . . 60
4.2.4 Creazione del dataset degli oggetti . . . . . . 61
4.2.5 Creazione del dataset di immagini base . . . . . . 62
4.2.6 Creazione del dataset utilizzato come riferimento . . . . . . 63
4.3 Il dataloader . . . . . . 63
4.3.1 L'oggetto Augmentor . . . . . . 63
4.3.2 Gli oggetti JsonLineObjectDataset e JsonLineMaskObjectDataset . . . . 65
4.3.3 L'oggetto ObjectDataloader e ShapeObjectDataloader . . . . . . 66
4.3.4 L'oggetto ImageFolder . . . . . . 66
4.3.5 Il Noise Generator . . . . . . 67
4.3.6 L'oggetto CoiganSeverstalSteelDefectsDataset . . . . . . 69
4.4 La pipeline di addestramento . . . . . . 70
4.4.1 Il generatore . . . . . . 70
4.4.2 La perceptual loss smooth masked . . . . . . 74
4.4.3 Il discriminatore dei difetti . . . . . . 76
4.4.4 Il discriminatore di riferimento . . . . . . 78
4.5 La valutazione del modello . . . . . . 79
4.5.1 Fréchet Inception Distance (FID) . . . . . . 80
4.5.2 I risultati . . . . . . 81

**5 Conclusioni e sviluppi futuri** **83**
5.1 Sviluppi futuri . . . . . . 83

**Elenco delle figure** **85**

**Bibliografia** **91**

# Capitolo 1

# Introduzione

## 1.1 Motivazione

### 1.1.1 Modelli neurali per la segmentazione nel controllo qualità

Oggi le reti neurali trovano un vasto impiego in moltissimi campi, dall'industria alla medicina, fino alla vita di tutti i giorni. Il grande vantaggio che ci portano è la capacità di apprendere da un insieme di dati, e di generalizzare su di uno nuovo, permettendoci di risolvere problemi che altrimenti sarebbero matematicamente troppo complessi da risolvere con un algoritmo. Ci sono vari esempi in cui i modelli neurali raggiungono risultati superiori a quelli ottenuti dall'uomo, in determinati task, o almeno se non lo superano in termini di accuratezza, lo fanno in termini di velocità, scalabilità, costi e prestazioni.

Un task in cui le reti neurali eccellono è la segmentazione di immagini, ovvero la classificazione pixel per pixel di un'immagine [23], questo tipo di task è utilizzato ad esempio nel campo medico per la segmentazione di organi [20], tumori [13], o in campo industriale per la segmentazione di difetti [30], per la verifica automatica della qualità di un prodotto o di un semilavorato.

Nel caso specifico, per la segmentazione dei difetti l'utilizzo di questo tipo di modelli è molto diffuso [12, 32], in quanto risolve un grave problema che affligge i reparti controllo qualità delle aziende, ovvero il calo della concentrazione al quale un operatore è soggetto dopo un certo numero di ore di lavoro. Infatti una persona per quanto allenata e preparata, dopo un certo numero di ore di lavoro, è soggetta a stanchezza e con essa la sua accuratezza nel riconoscere un difetto diminuisce, mentre un modello neurale adeguatamente addestrato, in condizioni ambientali stabili, come ad esempio una adeguata illuminazione, una videocamera ad alta risoluzione e un'adeguata distanza dal soggetto, sarà in grado di mantenere un'accuratezza costante, senza necessità di fermarsi per riposare. Questo si traduce in un risparmio di tempo e di denaro per l'azienda, In quanto il controllo manuale richiede più tempo ed è più soggetto ad errori, i quali spesso si trasformano in ritardi nella consegna dei prodotti, spese di trasporto aggiuntive per il ritorno o la sostituzione del prodotto, o addirittura la perdita di un cliente.

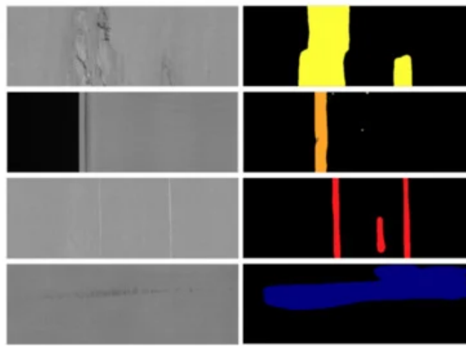


Figura 1.1: Un esempio preso dal **Severstal steel defect dataset** di difetti segmentati. credits: Neven Robby and Goedemé Toon, 2021, A Multi-Branch U-Net for Steel Surface Defect Type and Severity Segmentation. https://www.mdpi.com/2075-4701/11/6/870

### 1.1.2 Il dataset, requisiti e problematiche di realizzazione

La problematica di avere un modello con elevata accuratezza per task di segmentazione è relativa alla quantità e qualità dei dati necessari, i quali raramente sono disponibili opensource o per l'acquisto, rendendo necessaria la creazione di un dataset apposito. Molti task richiedono una grande quantità di dati per essere generalizzati correttamente, e ogni singolo esempio può richiede molta concentrazione da parte dell'annotatore in quanto non sempre i difetti sono ben visibili, ciò rende questa operazione soggetta ad errori. Per tali ragioni la realizzazione di un dataset può essere un'operazione molto complessa da gestire e portare a termine, ottenendo un risultato di qualità. Infatti vi sono diversi step che si devono seguire:

- **Acquisizione delle immagini**: Le immagini devono essere acquisite in modo da avere una buona qualità, in termini di risoluzione, messa a fuoco, illuminazione ecc. Se possibile in oltre dovrebbero avere una adeguata uniformità di condizioni (luce, distanza, ...) per garantire le migliori prestazioni da parte del modello, ovviamente solo se poi è possibile garantire le stesse condizioni anche nell'utilizzo finale del modello, altrimenti una grande varietà delle condizioni è preferibile.
- **Definizione delle classi**: Nel caso di dataset multi-classe, uno step molto importante è quello di scegliere accuratamente le classi e definire in maniera univoca l'associazione tra una classe e una particolare tipologia di difetto. Questo passaggio potrebbe sembrare banale ma in realtà nasconde delle grandi insidie, infatti una classificazione non adeguata andrà a causare confusione nel modello, diminuendo la sua accuratezza e/o rendendo il lavoro più difficile per gli annotatori andando a rallentare il processo di annotazione o comunque a ridurne la qualità. Questo tipo di problematiche purtroppo si manifestano chiaramente soltanto in uno stato avanzato del progetto, rendendo necessarie revisioni della documentazione, modifica di tutti gli esempi già annotati, con conseguente perdita di tempo e denaro.
- **Definizione della documentazione**: Questo passaggio è un'estensione del precedente, e consiste nella definizione di una documentazione Che specifichi senza ambiguità, ad un nuovo annotatore come riconoscere senza dubbio un difetto e classificarlo nella giusta classe. Questa fase spesso non termina prima dell'inizio dell'annotazione, ma si protrae

per tutta la durata del progetto, in quanto spesso nuovi casi non previsti si presentano durante l'annotazione, e la documentazione deve essere aggiornata in tempo reale.

- **Annotazione**: Questo è il passaggio più lungo e costoso, in quanto richiede una squadra di persone, che devono essere formate per lo specifico task, e che devono essere costantemente seguite per garantire la qualità del lavoro.

- **Revisione**: Assieme all'annotazione questo è un passaggio chiave, in quanto permette di verificare che l'annotazione sia stata fatta correttamente, e che non ci siano errori nell'annotazione. Spesso infatti gli annotatori acquisiscono dei bias errati nei confronti di una certa classe, o di un certo tipo di difetto, che deve essere identificato e reso noto all'annotatore per correggerlo, ed evitare che questo errore si ripeta in futuro. Per evitare che ciò accada oltre al primo annotatore lo stesso esempio viene solitamente rivisto da 2 o 4 persone diverse. Si noti che gli errori degli annotatori che non vengono identificati verranno appresi dal modello finale come una corretta classificazione, ciò giustifica un tale dispendio di risorse in questa fase.

### 1.1.3 Approccio al problema

La creazione di un dataset come precedentemente illustrato è un processo complesso e dispendioso, che richiede molte risorse umane e finanziarie, dunque l'intento di questo lavoro di tesi è di proporre un approccio alternativo che sia in grado di ridurre per quanto possibile la durata e il costo di questo lavoro. Partendo dal presupposto che almeno in parte il dataset deve essere realizzato manualmente, la proposta è quella di realizzare una certa quantità di campioni manualmente seguendo lo schema già visto, per poi addestrare un modello neurale per generare ulteriori esempi sintetici, raggiungendo un numero di esempi totali che permetta di addestrare un modello con buone prestazioni, ad un costo ridotto rispetto al caso in cui tutti i dati fossero stati realizzati manualmente.

Per la definizione della pipeline di generazione dei dati, si è partiti dal concetto di *generative adversarial network* (GAN), che è una tecnica di *machine learning* che permette di generare dati sintetici utilizzando come base dati reali, tali dati sintetici possono essere utilizzati per addestrare un modello neurale. Tale tecnica ha trovato riscontri positivi in molte ricerche pubblicate in ambito di *computer vision* [6], in cui i modelli GAN vengono utilizzati per espandere il numero di immagini presenti in un dataset e migliorare la generalizzazione di un modello di classificazione. Ovviamente gli aumenti di accuratezza, precisione e *recall* dipendono dal numero di esempi presenti nel dataset e dalla complessità del problema. Tale tecnica potrebbe essere considerata una versione più sofisticata di data augmentation, in quanto permette di generare dati sintetici molto più complessi e realistici di quelli che si possono ottenere con semplici trasformazioni geometriche o matematiche. Per generare dati utilizzabili per addestrare un modello di segmentazione però è necessario risolvere un'ulteriore problema, infatti un normale modello GAN, fedele alla sua definizione originale [11] , è in grado di generare intere immagini, che possono essere utilizzate per addestrare un modello di classificazione, ma non sono utilizzabili per addestrare un modello di segmentazione, in quanto per l'addestramento di tale architettura è necessario che gli oggetti di interesse abbiano una maschera che specifichi

la loro posizione. Ci sono vari approcci di *augmentation* per la segmentazione che risultano molto più semplici di addestrare un modello GAN, come il caso del metodo "*copy paste*" [9], il quale propone come *augmentation* per i dataset di segmentazione la copia di un oggetto presente in un'immagine, ritagliandolo attraverso la sua maschera, e incollandolo su di un nuovo background potenzialmente in una nuova posizione, tale metodo risulta estremamente efficacie per oggetti indipendenti dal contesto con contorni ben definiti, ma risulta inutile nel momento in cui l'oggetto che vogliamo generare ha una interdipendenza forte con l'area immediatamente circostante, pensiamo ad esempio un difetto su di un'auto, un graffio o una bozza, non potrà essere copiato da un'auto e incollato su di un'altra in quanto subentreranno una serie di di *artifacts*, come la variazione netta di colore tra l'auto e il difetto, rischiando di introdurre un *bias* nel modello, il quale finirebbe per cercare la variazione netta di colore e non più le *features* del difetto. Per risolvere questo problema con questa particolare categoria di dataset ci sono 2 principali strade illustrate di seguito.

### Generatore con architettura a solo decoder

Questo approccio prevede un'architettura a solo *decoder*, ovvero un modello che prende in ingresso un tensore di determinate dimensioni e che attraverso una serie di operazioni di *upsampling* o *dilated convolution* ad esempio, effettua un'espansione di tale tensore portandolo alle dimensioni finali. Generalmente si mette in ingresso un vettore casuale di dimensione definita, ottenendo in uscita un tensore delle dimensioni di un'immagine con i canali RGB ed eventualmente altri n canali per la maschere che identificano le classi desiderate. Un esempio di tale architettura è illustrata di seguito (Figura: 1.2).

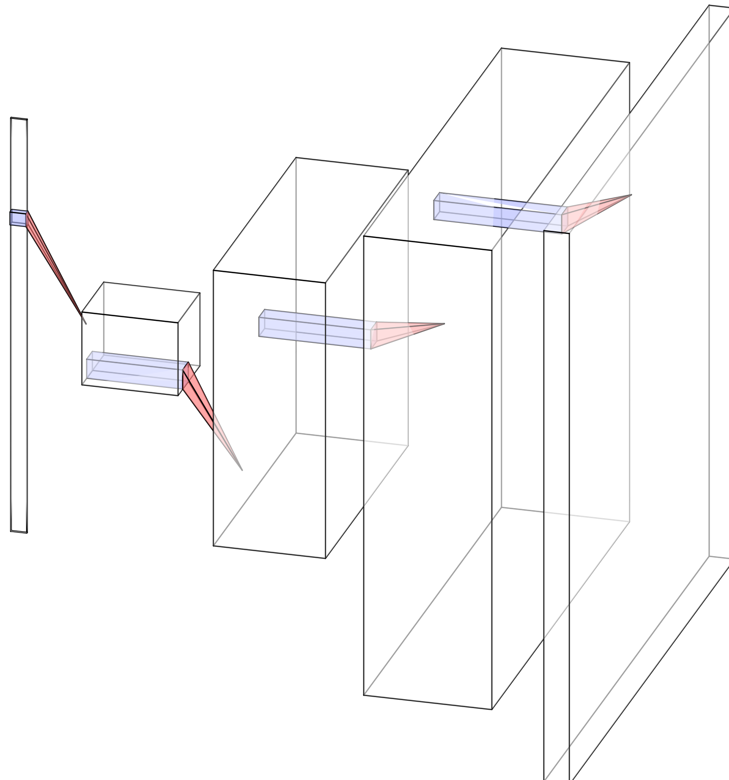


Figura 1.2: Esempio di architettura a solo decoder.

Questo approccio risulta più semplice da implementare, lasciando però al modello il compito di imparare a generare correttamente le immagini e delle maschere coerenti, compito non facile, che a seconda del task può necessitare di un elevato numero di esempi. Questa architettura dovrà imparare oltre alla struttura degli oggetti target, a posizionarli nell'immagine e a generare lo sfondo. Un'altra problematica di questo approccio è il controllo, infatti l'unico modo di interagire con tale modello è modificando il valore del vettore z dato in input, il quale permette di spostarsi nello spazio latente, al quale il modello associa diverse caratteristiche dell'immagine di output in maniera altamente non lineare, rendendo un eventuale controllo dell'output del modello molto difficile. La difficoltà di controllare il modello rende dunque difficoltoso o impossibile controllare, qualora fosse necessario, la posizione, l'intensità, la dimensione o la forma degli oggetti generati.

**Generatore con architettura a encoder-decoder**

Quest'ultimo è l'approccio scelto in questo progetto, in quanto permette di avere un maggiore controllo sull'output del modello, anche se prevede una training pipeline più complessa da gestire. Al contrario del caso precedente Il modello con struttura *encoder-decoder* (Figura: 1.3) permette di passare in ingresso un'immagine base e una o più maschere che identificano le aree dove determinati oggetti devono essere generati, trasformando il task di generazione puro in un task di *inpainting*. I vantaggi principali di questa tecnica stanno nel fatto che il modello non deve più apprendere la distribuzione degli oggetti nello spazio dell'immagine, ne deve apprendere in maniera troppo approfondita i background, ma si può focalizzare maggiormente sulla struttura degli oggetti da generare.

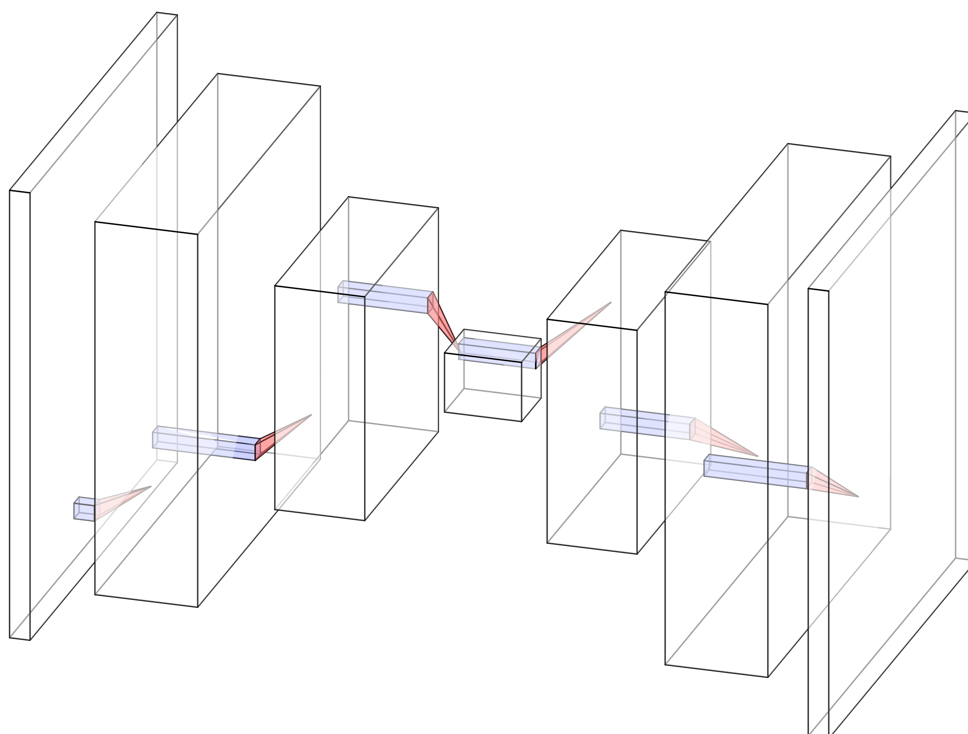


Figura 1.3: Esempio di architettura con encoder e decoder.

## 1.2 La nascita del deep learning

In questa sezione si darà un'idea generale di cos'è il deep learning e di come questo campo di ricerca sia nato dallo studio della neuroscienza e del machine learning.

### 1.2.1 Dal machine learning al deep learning

L'intelligenza artificiale è un campo di ricerca con l'obiettivo di risolvere una grande varietà di problemi, che per essere risolti attraverso la programmazione classica avrebbero bisogno di una grande quantità di conoscenze non disponibili, o semplicemente di troppo lavoro.
I programmi basati sul paradigma dell'intelligenza artificiale si propongono di superare questi ostacoli acquisendo direttamente queste conoscenze dai dati grezzi, tale capacità è nota come machine learning. Sotto questa grande famiglia di algoritmi si trovano altri sottogruppi quali il representation learning e all'interno di quest'ultimo il deep learning.
Il deep learning rispetto ai metodi più classici, tipicamente in grado di riconoscere soltanto relazioni lineari (come ad esempio l'SVM o support vector machine), si propone come alternativa per l'apprendimento di funzioni non lineari anche molto complesse. Il termine "deep learning" deriva proprio dalla capacità di riuscire a cogliere queste relazioni molto "profonde" tra i dati di ingresso e uscita.

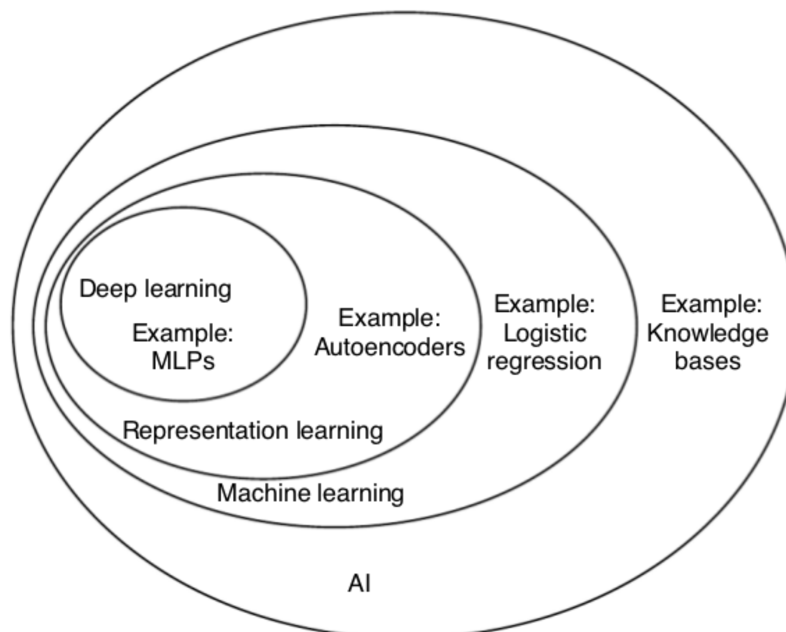



Figura 1.4: Un diagramma di ven che illustra le relazioni tra i diversi sottogruppi dell'intelligenza artificiale, vediamo infatti come il deep learning sia un sottogruppo del representation learning, che a sua volta è un sottogruppo del machine learning. credits: Yoshua Bengio, Ian J. Goodfellow, Aaron Courville 2015, From the book "Deep Learning"

### 1.2.2 Le reti neurali biologiche

Le reti neurali come concetto matematico fecero la loro prima comparsa in un articolo del 1957, pubblicato da Warren McCulloch e Walter Pitts "A logical calculus of the ideas immanent in nervous activity", articolo che gettò le basi per la costruzione di reti neurali artificiali come le conosciamo oggi partendo proprio dal sistema nervoso. Infatti le reti neurali artificiali sono ispirate alle reti neurali biologiche, le quali hanno un comportamento più complesso della controparte artificiale, in quanto le reti neurali artificiali devono fare i conti con la complessità computazionale che deve essere ridotta per garantire una elaborazione efficiente nei calcolatori. Il neurone biologico componente principale del cervello e del sistema nervoso, è costituito da un corpo cellulare o soma, dai dendriti, dall'assone e dalle sinapsi.

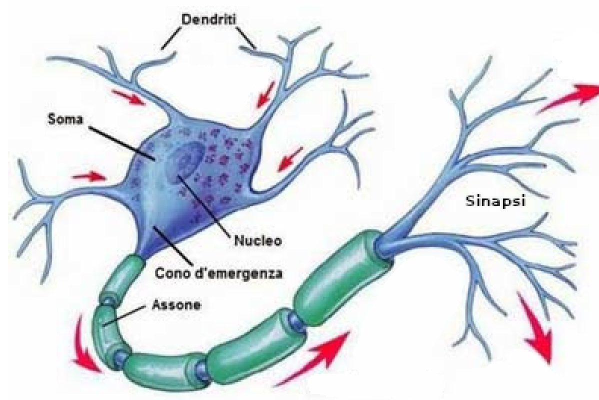



Figura 1.5: Schema di un neurone biologico.

Il neurone riceve degli impulsi da altri neuroni o da altri apparati sensoriali attraverso i dendriti, che sono delle strutture ramificate che si estendono dalla cellula, questi impulsi vengono accumulati all'interno del soma, e se la somma supera un certo valore di soglia si innesca la propagazione di un impulso, che viene trasmesso attraverso l'assone verso altri neuroni o verso altri organi. L'assone è una struttura che si estende dalla cellula, a seconda della tipologia di neurone può estendersi da pochi micrometri fino anche ad un metro, e presenta all'estremità opposta del soma le sinapsi, le quali consentono la propagazione dell'impulso dall'assone ad altri neuroni. La struttura dell'assone è rivestita dalla guaina mielinica che ne facilita la conduzione degli impulsi, maggiore è lo spessore della guaina minore è la resistenza al passaggio dell'impulso, e dunque maggiore sarà l'ampiezza del segnale in uscita a parità di quello di ingresso. Tale meccanismo è utilizzato per accumulare informazione nella struttura della rete neurale biologica.

## 1.3 Le reti neurali feed forward

Uno dei primi modelli ad essere proposti e utilizzati nella pratica è stato quello delle reti neurali feedforward (o multi layer perceptron MLP), in cui i neuroni sono disposti in strati, e l'output di ogni neurone di uno strato è connesso con l'input di tutti i neuroni dello strato successivo, attraverso delle connessioni che conservano un peso, la configurazione di tali pesi determina il comportamento della rete. Tali reti come dice il nome propagano l'informazione dallo strato di

input a quello di output attraverso i layer intermedi, in modo lineare, senza retropropagazioni intermedie dell'informazione.

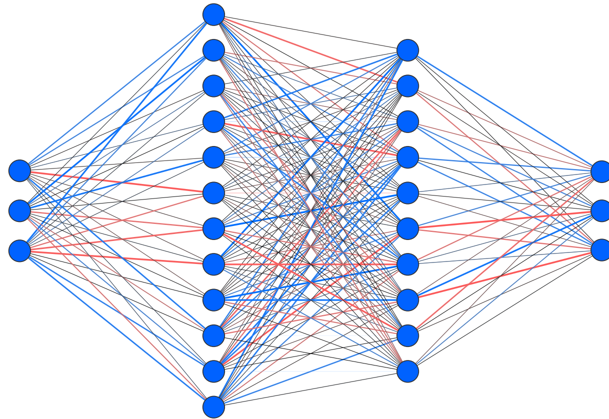


Figura 1.6: Esempio di rete neurale feedforward. In tale rete è possibile vedere i neuroni rappresentati dai nodi del grafo, e le interconnessioni tra di essi che definiscono il peso di ogni relazione, con in rosso un peso positivo e in blu un peso negativo, l'intensità del colore indica la forza della relazione.

### 1.3.1 Il neurone artificiale

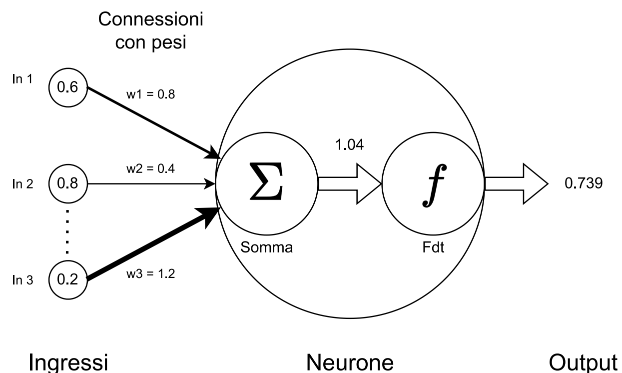


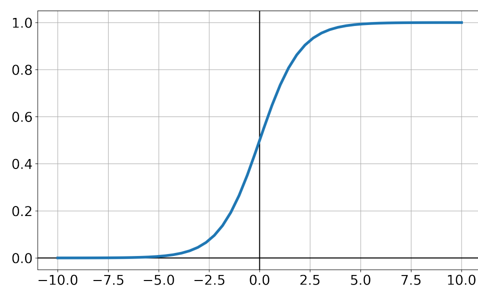

Figura 1.7: Esempio di Neurone artificiale.

Il neurone artificiale emula il comportamento del neurone biologico, semplificandone notevolmente la complessità, una delle più importanti semplificazioni è che il neurone artificiale opera in un regime temporale discreto e non continuo come la controparte. Il neurone artificiale inoltre non utilizza un meccanismo di accumulazione e spike, ma restituisce un output per ogni input ricevuto, ciò che varia è l'intensità di questo output, che dipende dall'intensità degli input ricevuti, e dai pesi delle connessioni con tali input. Il neurone artificiale inoltre è provvisto di una funzione di trasferimento che mappa la somma pesata degli input ricevuti con l'uscita. Vediamo dunque l'espressione che caratterizza il comportamento di un neurone artificiale:

$$y = f(P) = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b) = f(\sum_{i=1}^{n} w_i x_i + b) \tag{1.1}$$

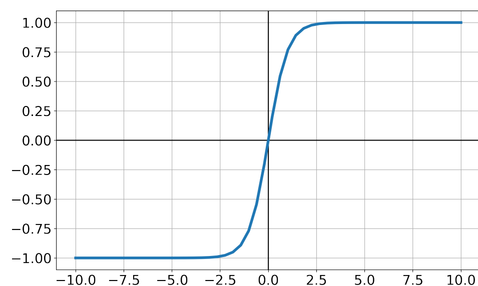
Considerando $y$ l'output del neurone, $\vec{\mathbf{w}}$ il vettore dei pesi delle connessioni in ingresso, $\vec{\mathbf{x}}$ il vettore degli input, $b$ il bias ovvero un valore aggiunto alla somma pesata degli input dipendente dal neurone e $f$ la funzione di trasferimento. La funzione di attivazione o funzione di trasferimento del neurone è la componente che conferisce alla rete la capacità di generare degli output che hanno una relazione non lineare rispetto agli input ricevuti, e dunque che gli permette di apprendere funzioni non lineari. La funzione di attivazione solitamente deve essere derivabile, o almeno derivabile a tratti per poter essere utilizzata in un contesto di apprendimento, in

quanto la derivata della funzione di attivazione viene utilizzata per calcolare il gradiente della funzione di errore dall'algoritmo Backpropagation, e in seguito per aggiornare i pesi da parte della funzione di ottimizzazione (es. SGD). Di seguito sono mostrate alcune delle più comuni funzioni di attivazione:



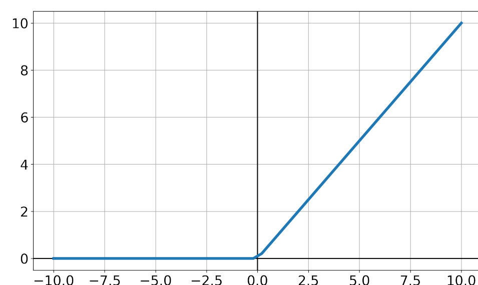

(a) Sigmoide

$$\mathbf{f(x)} = \frac{\mathbf{1}}{\mathbf{1+e^{-x}}}$$



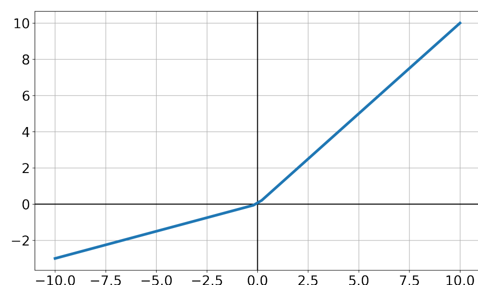

(b) Tangente iperbolica

$$\mathbf{f(x)} = \frac{\mathbf{e^{x}-e^{-x}}}{\mathbf{e^{x}+e^{-x}}}$$



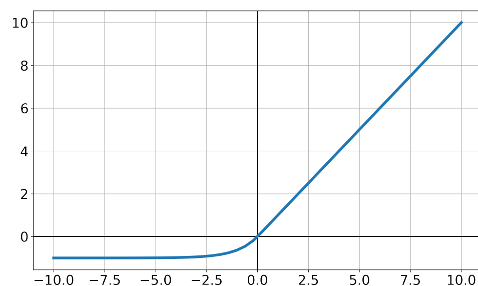

(e) Rectified linear unit (RELU)

$$\mathbf{f(x)} = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$



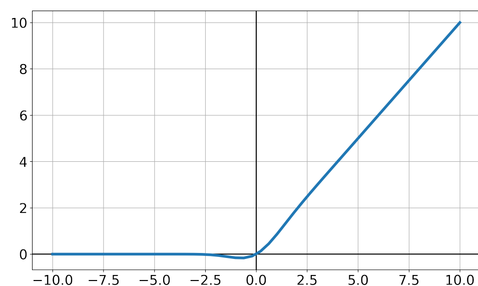

(f) Leaky RELU

$$\mathbf{f(x)} = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases}$$




(i) Exponential linear unit (ELU)

$$\mathbf{f(x)} = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^{x}-1) & \text{if } x < 0 \end{cases}$$




(j) Gaussian error linear unit (GELU)

$$\mathbf{f(x)} = \mathbf{x} * \mathbf{\Phi(x)}$$

La sigmoide e la tangente iperbolica, sono funzioni molto vecchie utilizzate sin dai primi anni '90, ma sono state sostituite da funzioni più moderne e semplici da calcolare, come le funzioni RELU e Leaky RELU. Vengono ancora utilizzate però in casi particolari come nello stadio finale di una rete neurale, quando si ha bisogno di un output che sia compreso tra 0 e 1. Le funzioni ELU e GELU sono invece funzioni di attivazione più recenti, che vengono utilizzate per migliorare l'apprendimento delle reti neurali in casi specifici, ma in generale RELU e Leaky RELU sono le funzioni più utilizzate, in quanto offrono un buon compromesso tra prestazioni e accuratezza.

### 1.3.2 Il Back-propagation

La prima volta che questo algoritmo fu proposto fu nel 1986 da David E. Rumelhart, Geoffry E. Hinton e Ronald J. Williams, su nature con l'articolo *Learning representations by back-propagating errors*. Questo algoritmo ha segnato da quel momento una vera e propria rivoluzione nel campo dell'apprendimento automatico, rendendo possibile l'addestramento dei modelli neurali come li conosciamo oggi e plasmando lo scenario attuale dell'intelligenza artificiale.

Ciò che questo algoritmo fa effettivamente è calcolare il gradiente della funzione di errore nello spazio dei pesi di una rete neurale, partendo dal layer di uscita e andando a calcolare il gradiente per ogni layer precedente fino ad arrivare al layer di input, tale gradiente può essere poi utilizzato per aggiornare i pesi della rete attraverso una funzione di ottimizzazione, in modo da minimizzare o massimizzare la funzione di costo che si vuole ottimizzare.

Da un punto di vista matematico, data la precedente definizione di neurone e rete neurale, possiamo definire una rete neurale come una funzione $\mathbf{g(x)}$ come combinazione di composizione di funzioni e moltiplicazioni di matrici.

$$\mathbf{\tilde{y}} = \mathbf{g(\tilde{x})} = \mathbf{f_L(W^L \cdot f^{L-1}(W^{L-1} \cdot f^{L-2}(\ldots W^3 \cdot f^2(W^2 \cdot f^1(W^1 \cdot \tilde{x}))\ldots)))} \tag{1.2}$$

Dove $\mathbf{f_L}$ è la funzione di attivazione del layer di uscita, $\mathbf{f_i}$ è la funzione di attivazione del layer $i$ e $\mathbf{W^i}$ è la matrice dei pesi del layer $i$. Abbiamo inoltre che $\mathbf{\tilde{x}}$ e $\mathbf{\tilde{y}}$ sono rispettivamente il vettore di input e il vettore di output della rete neurale, se consideriamo $\mathbf{\hat{y}}$ il vettore di output desiderato, possiamo definire la funzione di costo. In questo caso utilizzeremo la loss MSE (Mean Squared Error), ma si può sostituire con qualsiasi funzione di costo.

$$\mathbf{E} = \mathbf{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{1.3}$$

A questo punto data una determinata coppia di input e output $(\mathbf{\tilde{x}}, \mathbf{\tilde{y}})$, possiamo calcolare il gradiente della funzione di costo attraverso la *regola della catena*, che ci permette di calcolare il gradiente della funzione di costo per ogni peso della rete.

$$\frac{\partial \mathbf{E}}{\partial \mathbf{w^l_{ij}}} = \frac{\partial \mathbf{E}}{\partial \mathbf{a^l}} \cdot \frac{\partial \mathbf{a^l}}{\partial \mathbf{z^l}} \cdot \frac{\partial \mathbf{z^l}}{\partial \mathbf{w^l_{ij}}} \tag{1.4}$$

Otteniamo così il gradiente della funzione di costo per il peso $w^l_{ij}$ appartenente al layer $l$, al neurone $i$ e alla sinapsi $j$.

Per l'esecuzione dell'algoritmo back propagation è necessario effettuare il caching dei valori intermedi calcolati durante il passaggio in avanti, nello specifico degli input pesati dei neuroni prima della funzione di attivazione $\mathbf{\tilde{z}^l}$ e l'output dei neuroni dopo la funzione di attivazione $\mathbf{\tilde{a}^l}$ per ogni layer.

Consideriamo la derivata della funzione di errore rispetto all'input del modello:

$$\frac{\partial \mathbf{E}}{\partial \mathbf{x}} = (\frac{\partial \mathbf{E}}{\partial \mathbf{a^L}}) \circ (\frac{\partial \mathbf{a^L}}{\partial \mathbf{z^L}} \cdot \frac{\partial \mathbf{z^L}}{\partial \mathbf{a^{L-1}}}) \circ (\frac{\partial \mathbf{a^{L-1}}}{\partial \mathbf{z^{L-1}}} \cdot \frac{\partial \mathbf{z^{L-1}}}{\partial \mathbf{a^{L-2}}}) \circ \cdots \circ (\frac{\partial \mathbf{a^2}}{\partial \mathbf{z^2}} \cdot \frac{\partial \mathbf{z^2}}{\partial \mathbf{a^1}}) \circ (\frac{\partial \mathbf{a^1}}{\partial \mathbf{z^1}} \cdot \frac{\partial \mathbf{z^1}}{\partial \mathbf{x}}) \tag{1.5}$$

Dove $\circ$ è il prodotto di Hadamard, un semplice prodotto element-wise tra matrici di dimensioni uguali, che moltiplica elemento per elemento le due matrici. Osservando questa formulazione della derivata possiamo notare che $\frac{\partial \mathbf{a^L}}{\partial \mathbf{z^L}} \cdot \frac{\partial \mathbf{z^L}}{\partial \mathbf{a^{L-1}}}$ è la derivata della funzione di attivazione moltiplicata per la matrice dei pesi del layer stesso. Inoltre possiamo riscrivere la derivata $\frac{\partial \mathbf{E}}{\partial \mathbf{a^L}}$ in termini di gradiente $\nabla_{\mathbf{a}}\mathbf{E}$, invertendo l'ordine dei prodotti e trasponendo le matrici:

$$\nabla_{\mathbf{x}}\mathbf{E} = (\mathbf{W^1})^{\mathbf{T}} \cdot (\mathbf{f^1})' \circ \cdots \circ (\mathbf{W^{L-1}})^{\mathbf{T}} \cdot (\mathbf{f^{L-1}})' \circ (\mathbf{W^L})^{\mathbf{T}} \cdot (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E} \tag{1.6}$$

A questo punto possiamo introdurre i prodotti parziali del gradiente per determinare il gradiente della funzione di errore ad un determinato layer $l$:

$$\delta^{\mathbf{l}} = (\mathbf{f^l})' \circ (\mathbf{W^{l+1}})^{\mathbf{T}} \cdot (\mathbf{f^{l+1}})' \circ \cdots \circ (\mathbf{W^L})^{\mathbf{T}} \cdot (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E} \tag{1.7}$$

E notiamo che ogni prodotto parziale può essere definito come il prodotto tra il gradiente e la matrice trasposta dei pesi del layer successivo per la derivata della funzione di attivazione del layer stesso.

$$\delta^{\mathbf{l-1}} = (\mathbf{f^{l-1}})' \circ (\mathbf{W^l})^{\mathbf{T}} \cdot \delta^{\mathbf{l}} \tag{1.8}$$

Quindi:

$\delta^{\mathbf{L}} = (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E}$

$\delta^{\mathbf{L-1}} = (\mathbf{f^{L-1}})' \circ (\mathbf{W^L})^{\mathbf{T}} \cdot \delta^{\mathbf{L}} = (\mathbf{f^{L-1}})' \circ (\mathbf{W^L})^{\mathbf{T}} \cdot (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E}$

$\dots$

$\delta^{\mathbf{2}} = (\mathbf{f^2})' \circ (\mathbf{W^3})^{\mathbf{T}} \cdot \delta^{\mathbf{3}} = (\mathbf{f^2})' \circ (\mathbf{W^3})^{\mathbf{T}} \cdot \dots \circ (\mathbf{W^L})^{\mathbf{T}} \cdot (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E}$

$\delta^{\mathbf{1}} = (\mathbf{f^1})' \circ (\mathbf{W^2})^{\mathbf{T}} \cdot \delta^{\mathbf{2}} = (\mathbf{f^1})' \circ (\mathbf{W^2})^{\mathbf{T}} \cdot (\mathbf{f^2})' \circ (\mathbf{W^3})^{\mathbf{T}} \cdot \dots \circ (\mathbf{W^L})^{\mathbf{T}} \cdot (\mathbf{f^L})' \circ \nabla_{\mathbf{a^L}}\mathbf{E}$

In tal modo possiamo calcolare i prodotti parziali del gradiente per ogni layer, partendo dallo strato di output all'indietro, minimizzando così il numero di operazioni richieste per il calcolo del gradiente. Per ottenere i gradienti dei pesi è sufficiente moltiplicare il prodotto parziale del gradiente per l'output del layer precedente:

$$\nabla_{\mathbf{w^l}}\mathbf{E} = \delta^{\mathbf{l}} \cdot (\mathbf{a^{l-1}})^{\mathbf{T}} \tag{1.9}$$

L'oggetto $\nabla_{\mathbf{w^l}}\mathbf{E}$ rappresenta una matrice della stessa dimensione della matrice dei pesi del layer $l$, contenente i gradienti della funzione di errore di tali pesi, mentre $a^{l-1}$ è l'output di tutte le unità del layer precedente.

Si consideri che è possibile scomporre un elemento della matrice $\nabla_{\mathbf{w^l}}\mathbf{E}$ nel seguente modo:

$$\nabla_{\mathbf{w^l}}\mathbf{E_{k,k}} = \frac{\partial \mathbf{E}}{\partial \mathbf{w^l_{k,k}}} \tag{1.10}$$

Tale gradiente a questo punto può essere utilizzato per aggiornare il peso $w^l_{k,k}$, applicando un semplice coefficiente di apprendimento $\eta$:

$$\mathbf{W^1_{k,k}} = \mathbf{W^1_{k,k}} + \mathbf{\Delta W^1_{k,k}} \tag{1.11}$$

$$\mathbf{\Delta W^1_{k,k}} = -\eta \cdot \nabla_{\mathbf{w^1}}\mathbf{E_{k,k}} \tag{1.12}$$

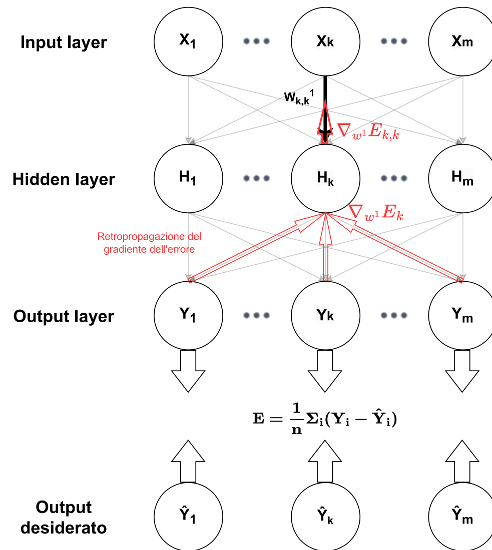



Figura 1.9: Esempio di retropropagazione su di una rete semplificata a 2 strati.

é importante notare che questo è un'approccio molto basilare, che può essere migliorato attraverso l'uso di tecniche di ottimizzazione più evolute come ad esempio l'ottimizzazione SGD (Stochastic Gradient Descent) o l'ottimizzazione ADAM.

### 1.3.3 Teorema di approssimazione universale

Le reti neurali artificiali, non hanno dimostrato solo nella pratica con risultati sperimentali la loro efficacia nel risolvere i problemi, ma anche nella teoria, infatti in molti studi è stata provata la loro efficacia come approssimatori universali. Quando si parla di approssimatori universali ci si riferisce a delle funzioni che possono approssimare su un dato intervallo di valori qualunque altra funzione continua. Nel caso particolare delle reti neurali vi è una grande quantità di varianti, per le quali si ha una diversa dimostrazione per tale proprietà.

Tra i più importanti nel 1989 George Cybenko in *Approximation by superpositions of a sigmoidal function* [4], dimostrò che la sovrapposizione di un numero finito di istanze di una singola funzione univariata può approssimare qualsiasi funzione continua di n variabili con supporto nell'ipercubo unitario. Questo risultato permette di asserire che ogni funzione continua può essere approssimata da una rete neurale MLP (feedforward) avente uno strato nascosto con un numero finito di unità, con come funzione di attivazione una funzione sigmoide arbitraria. Di seguito il relativo teorema:

**Teorema 1** *Sia sigma qualsiasi funzione sigmoide arbitraria. Le sommatorie finite della forma*

$$\mathbf{G(x)} = \sum_{\mathbf{i=1}}^{\mathbf{N}} \alpha \cdot \sigma(\mathbf{y_i^T x} + \mathbf{\Theta_i}) \tag{1.13}$$

*sono dense in $C(I_n)$. In altre parole, data qualunque $f \in C(I_n)$ e $\epsilon > 0$ esiste una somma $G(x)$ con la forma sopra descritta tale che:*

$$|\mathbf{G(x)} - \mathbf{f(x)}| < \epsilon \quad \forall \mathbf{x} \in \mathbf{I_n} \tag{1.14}$$

Con questo teorema Cybenko dimostra che una rete neurale MLP con un solo strato nascosto può approssimare qualsiasi funzione continua ma non è in grado di stabilire quante unità deve avere; in base alla complessità della funzione da approssimare il numero potrebbe essere molto grande. Altre ricerche negli anni seguenti hanno studiato molteplici varianti di questo teorema, ad esempio per neuroni con funzione di attivazione RELU o il caso con profondità del modello arbitraria.

## 1.4 Le reti neurali convoluzionali

Le reti neurali convoluzionali sono modelli concepiti per analizzare dati che hanno una struttura spaziale, e nei quali le relazioni tra i dati hanno relazioni locali molto forti, come nel caso delle immagini. La loro struttura è simile a quella che si trovano nel nervo ottico degli organismi viventi. Uno dei maggiori vantaggi che ha portato l'avvento delle CNN è stato quello di ridurre drasticamente il numero di parametri da apprendere, i quali con le comuni MLP crescevano esponenzialmente all'aumentare della risoluzione dell'immagine, rendendo di fatto il problema intrattabile. Sono dette anche *Shift invariant artificial neural networks* (SIANN) in quanto sono resistenti alla traslazione dell'input.

### 1.4.1 Storia delle CNN

La prima pubblicazione alla quale si deve l'invenzione delle CNN è stato un lavoro di David Hunter Hubel e Torsten Wiesel *"Receptive fields of single neurones in the cat's striate cortex"*, i quali nel 1959 hanno studiato la struttura della corteccia visiva del cervello di un gatto, scoprendo che tali neuroni avevano una struttura particolare, che li rendeva soggetti agli stimoli di una certa porzione di retina, detta *campo ricettivo*, tale area era regolare per tutti i neuroni e tutti avevano una certa sovrapposizione dei rispettivi campi ricettivi.

Il primo lavoro a sfruttare i concetti appresi da Hubel e Wiesel è stato quello di Kunihiko Fukushima il quale nel 1980 ha pubblicato il suo lavoro *Neocognitron* [7], un sistema di riconoscimento di immagini, utilizzato per la prima volta per il riconoscimento di numeri scritti a mano in giapponese Questa implementazione di modello neurale è molto vicino al modello naturale, utilizza infatti due tipologie di neuroni:

- **S-cells**: sono i neuroni che si occupano di estrarre le features locali, come ad esempio linee o bordi, presentando le caratteristiche delle cellule della corteccia visiva primaria, come il campo ricettivo limitato ad un'area ristretta.

- **C-cells**: sono i neuroni che si occupano di estrarre le features globali, infatti questi ricevono in input le uscite dei neuroni S-cells, e dunque lavorano con pattern più complessi, come ad esempio semi-cerchi o quadrati ecc. Anche in questo caso c'è una somiglianza con il modello biologico in quanto questi neuroni hanno un campo ricettivo molto più ampio, e costituiscono la corteccia visiva secondaria.

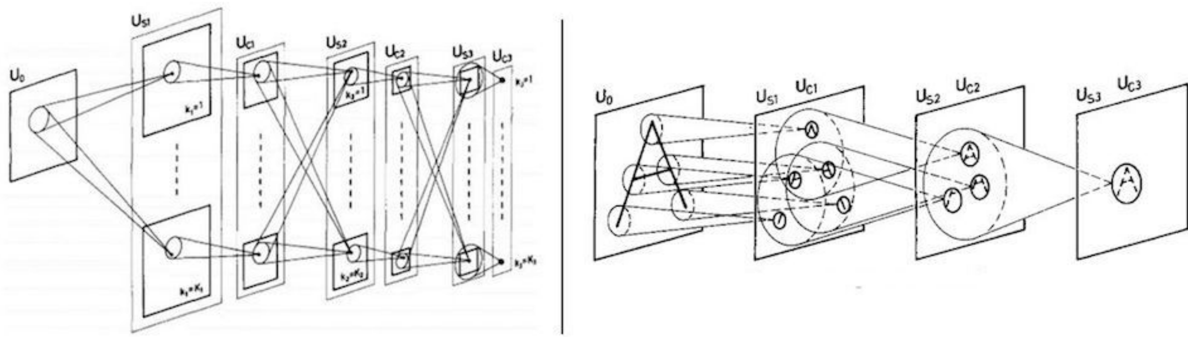



Figura 1.10: L'immagine raffigura schematicamente il funzionamento di Neocognitron.
credits: Kunihiko Fukushima 1980 [7].

Date queste caratteristiche, possiamo notare come il modello neurale di Fukushima sia molto simile a quello che si trova nel cervello ma ancora presenta delle problematiche, come ad esempio la mancanza di un algoritmo di addestramento end-to-end come il backpropagation e l'invarianza alla traslazione, in quanto ogni S-cell ha un campo ricettivo molto limitato e non condivide i suoi pattern con le altre, dunque un modello addestrato con questa struttura potrebbe non essere in grado di riconoscere un oggetto in ogni punto dell'immagine con la stessa precisione. Sempre a Fukushima si deve anche l'introduzione della funzione di trasferimento *RELU*, precedentemente discussa.

Seguì poi il lavoro di Alex Waibel *"Phoneme Recognition Using Time-Delay Neural Networks"* [29] che nel 1989 fece un'ulteriore passo verso le moderne CNN, introducendo per la prima volta l'invarianza alla traslazione, ma si parlava ancora di una convoluzione 1D. Questo lavoro però non era focalizzato sulle immagini ma sull'analisi del suono, in particolare sul riconoscimento di fonemi. Ma l'utilizzo di questo modello su due dimensioni, ovvero il tempo e la frequenza, lo rendeva concettualmente adatto anche all'analisi delle immagini. Un altro fatto interessante è che in questo lavoro viene introdotto il concetto di *pooling*, introdotto proprio per ridurre la dimensione dei tensori durante la propagazione nel modello, e ridurre così il numero di parametri da apprendere, soluzione oggi molto utilizzata nelle CNN moderne, in diverse varianti che poi verranno discusse. Nell'immagine che segue, tratta proprio dall'articolo citato si può vedere come questo modello effettuava la convoluzione 1D nel tempo ed il pooling nel tempo e nella frequenza, tra i vari *layer*.

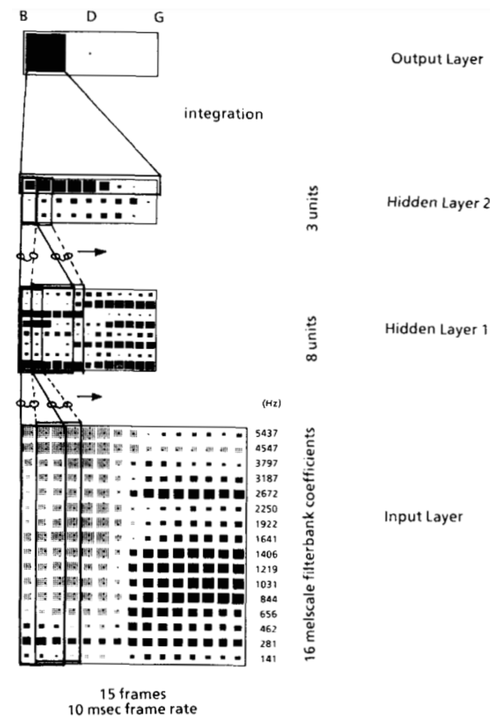



Figura 1.11: L'immagine tratta dallo stesso articolo illustra schematicamente l'architettura del modello TDNN.
credits: Alex Waibel et al. 1989 [29].

In fine una delle prime ricerche che si sono avvicinate di più all'implementazione moderna

delle CNN è stata quella di Yann LeCun et al. *"Backpropagation applied to handwritten zip code recognition"* [19] pubblicato nel 1989, in cui è stato utilizzato il backpropagation per l'addestramento di una rete convoluzionale per il riconoscimento di numeri scritti a mano.

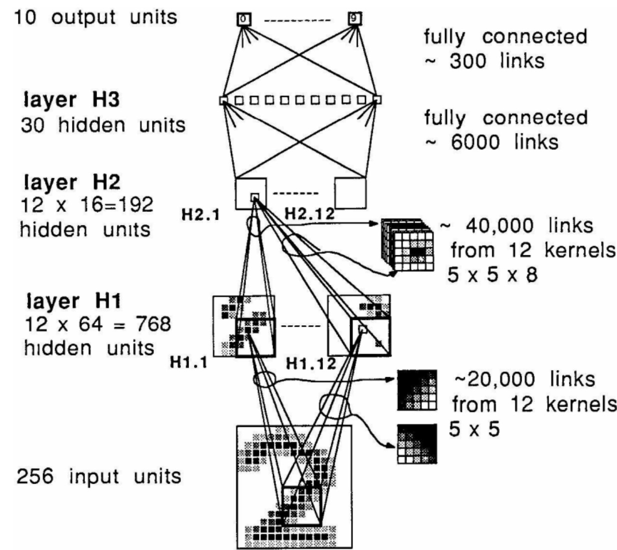



Figura 1.12: L'immagine tratta dal medesimo articolo illustra l'architettura proposta. credits: Yann LeCun et al. 1989 [19].

La rete proposta da LeCun in questa pubblicazione aveva un'architettura molto semplice, basta su 2 strati convoluzionali 2D, e 2 starti MLP, l'utilizzo del backpropagation per l'addestramento dei kernel di questi due starti convoluzionali, per quanto oggi sembri banale era un'innovazione di grande impatto, in quanto fino a quel momento i kernel dei filtri convoluzionali venivano definiti manualmente. Un'altra caratteristica interessante fu quella dell'utilizzo della tecnica del "weights sharing", la caratteristica che consente alle CNN di ridurre drasticamente il numero di parametri da apprendere, rispetto alle MLP o a precedenti implementazioni come Neocognitron.

### 1.4.2 La convoluzione

In questa sezione andremo a vedere come funziona la convoluzione, e come viene implementata nelle CNN moderne. partendo dalla definizione formale di convoluzione, che è la seguente:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{1.15}$$

dove $f$ e $g$ sono due funzioni, e $*$ è l'operatore di convoluzione. Questa definizione è valida per funzioni definite su un dominio continuo, ma spesso nella pratica si utilizza la convoluzione su un dominio discreto, come ad esempio le immagini, le quali sono definite da un numero finito di pixel. In questo caso la convoluzione si può definire come segue:

$$(f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) \tag{1.16}$$

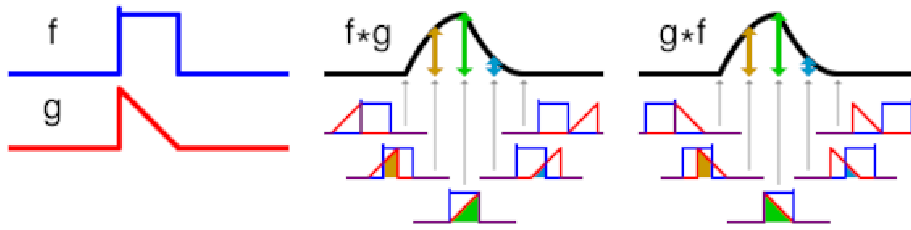



Figura 1.13: L'immagine illustra l'operazione di convoluzione.
credits: Wikipedia. https://en.wikipedia.org /wiki/Convolution

Per fare un esempio pratico la convoluzione equivale ad effettuare un prodotto tra due funzioni, ma con un'operazione di shift, che è il motivo per cui la convoluzione è anche detta operazione di correlazione. In generale si avrà come risultato una funzione che è più alta dove le due funzioni in ingresso sono più simili, e più bassa dove queste sono più diverse, o ancora negativa dove queste sono opposte. Come è possibile vedere nell'immagine 1.13, dove l'onda a dente di sega ha una sovrapposizione maggiore con l'onda quadra la loro convoluzione avrà un valore maggiore, inoltre è possibile notare come tale operazione sia commutativa.

La convoluzione discreta nelle immagini, rispetto al caso continuo tra funzioni, si avvale dell'uso di kernel, ovvero di matrici di dimensioni finite, che vengono spostate lungo l'immagine, moltiplicandole e sommando i valori in ogni posizione, in modo da produrre un'immagine di output che è la convoluzione dell'immagine di input con il kernel.

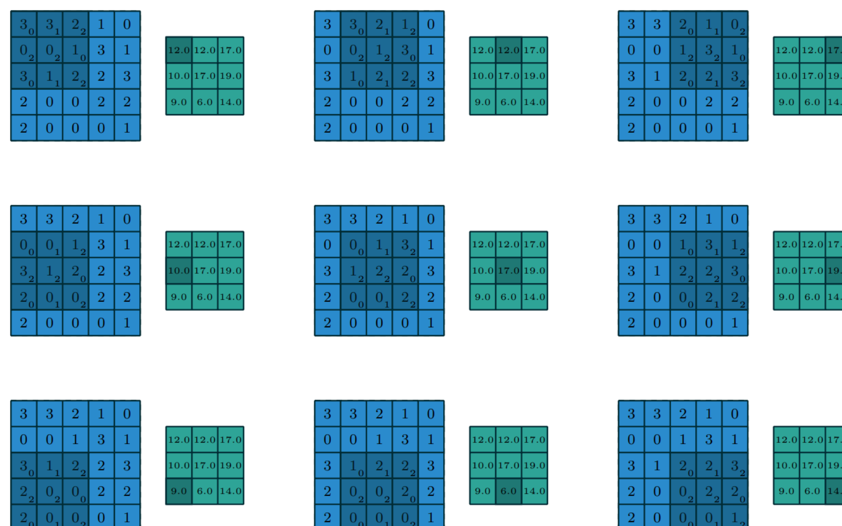


Figura 1.14: Operazione di convoluzione 2D discreta, in blu abbiamo una matrice 5x5 che rappresenta un'immagine, mentre in verde il risultato di una convoluzione, con un kernel 3x3. I valori del kernel sono raffigurati in basso a destra delle caselle interessate dalla convoluzione. credits: Dumoulin et al. 2016 [5].

Vediamo un esempio pratico, consideriamo due kernel 3x3, uno con un pattern verticale, e l'altro con un pattern orizzontale e un'immagine di input 5x5:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 \\ 3 & 4 & 5 & 4 & 3 \\ 2 & 3 & 4 & 3 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} \tag{1.17}$$

Vediamo la convoluzione di questi due kernel con la matrice di esempio 5x5:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 \\ 3 & 4 & 5 & 4 & 3 \\ 2 & 3 & 4 & 3 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -6.0 & 0.0 & 6.0 \\ -6.0 & 0.0 & 6.0 \\ -6.0 & 0.0 & 6.0 \end{bmatrix} \tag{1.18}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 \\ 3 & 4 & 5 & 4 & 3 \\ 2 & 3 & 4 & 3 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -6.0 & -6.0 & -6.0 \\ 0.0 & 0.0 & 0.0 \\ 6.0 & 6.0 & 6.0 \end{bmatrix} \tag{1.19}$$

In questo modo si ottiene un'immagine di output che ha valori più alti dove i pattern presenti nel kernel sono simili mentre si hanno valori negativi dove il pattern è opposto, nello specifico, possiamo vedere come nella convoluzione 1.18, ha un andamento decrescente verso destra, e infatti la convoluzione con l'immagine restituisce un valore elevato dove l'immagine ha un andamento decrescente verso destra. Vediamo nel dettaglio però quali operazioni vengono effettuate per ottenere l'immagine di output, considerando sempre la convoluzione 1.18, vediamo una lista di operazioni, una per ogni valore della matrice di output, che corrispondono alle moltiplicazioni e somme che vengono effettuate tra il kernel e l'immagine, in ogni posizione in cui il kernel si sovrappone all'immagine:

- $1 * 1 + 0 * 2 + -1 * 3 + 1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 = -6.0$
- $1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 = 0.0$
- $1 * 1 + 0 * 2 + -1 * 3 + 1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 = 6.0$
- $1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 = -6.0$
- $1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 + 1 * 1 + 0 * 2 + -1 * 3 = 0.0$
- $1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 = 6.0$
- $1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 + 1 * 1 + 0 * 2 + -1 * 3 = -6.0$
- $1 * 2 + 0 * 3 + -1 * 4 + 1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 = 0.0$
- $1 * 3 + 0 * 4 + -1 * 5 + 1 * 2 + 0 * 3 + -1 * 4 + 1 * 1 + 0 * 2 + -1 * 3 = 6.0$

Di seguito è riportata l'immagine 1.15 di Lena Forsén, un'attrice svedese la cui foto è usata frequentemente per esempi e test di algoritmi di computer vision, a cui sono stati applicati i due kernel visti prima, alla prima immagine è stato applicato il kernel 1.18 mentre alla seconda è stato applicato il kernel 1.19.

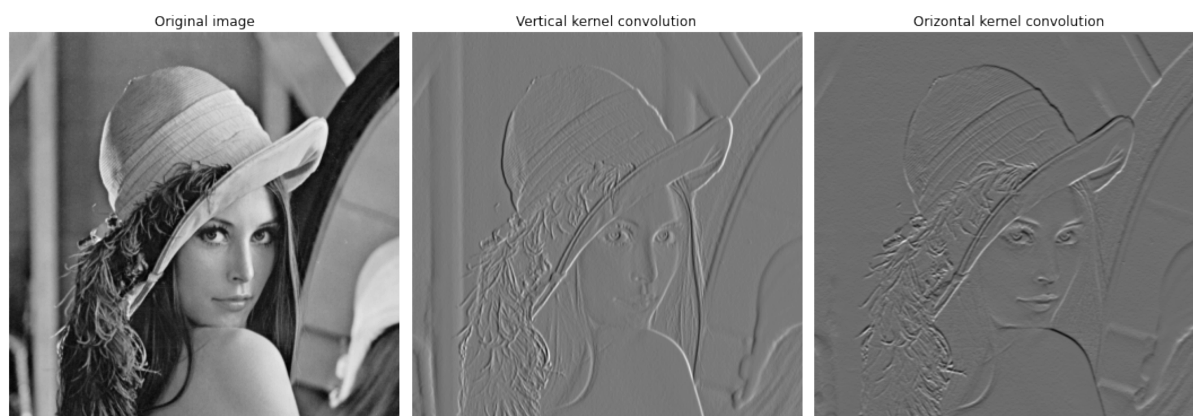



Figura 1.15: Immagine di Lena Forsén, prima e dopo la convoluzione con i due kernel, che amplificano i bordi verticali e orizzontali.

### 1.4.3 I parametri della convoluzione

In questa sezione andremo a vedere quali sono i principali parametri della convoluzione, e come questi la influenzano, vedremo in particolare: padding, stride, kernel size e numero di filtri.

#### Padding

Il padding è un parametro che viene utilizzato per aumentare la dimensione dell'immagine di ingresso prima di effettuare la convoluzione, e ottenere così una dimensione di output maggiore, questo è utile quando si vuole mantenere la dimensione dell'immagine di output uguale a quella di input, tale risultato si ottiene aggiungendo delle righe e colonne di valori intorno alla matrice di input originale. Ci sno due tipologie principali di padding:

- **Zero padding**: il padding viene aggiunto con dei valori pari a zero, questo è utile quando si vuole mantenere la dimensione dell'immagine di output uguale a quella di input.
- **Reflection padding**: il padding viene aggiunto con dei valori uguali ai valori dell'immagine di input, riflessi rispetto ai bordi.

#### Stride

Lo stride è un parametro che viene utilizzato per decidere di quanto si sposta il kernel durante la convoluzione, ad esempio con uno stride pari a 2, il kernel si sposterà di 2 pixel in orizzontale e/o in verticale, questo è utile quando si vuole ridurre la dimensione dell'immagine di output, con il giusto stride si può effettuare la convoluzione e ottenere ad esempio un'immagine di output di dimensione pari a $\frac{1}{2}$ dell'immagine di input.

### Kernel size

La kernel size o dimensione del kernel è un parametro che viene utilizzato per decidere la dimensione del kernel con cui effettuare la convoluzione. tipicamente i kernel sono quadrati, ma è possibile utilizzare anche kernel rettangolari, ad esempio un kernel di dimensione 3x5 o 1x7. Per riprendere il collegamento con la controparte biologica delle reti neurali, il kernel è l'equivalente del campo visivo del neurone, aumentando la dimensione del kernel aumentiamo il campo visivo del neurone, e quindi aumentiamo la capacità di riconoscere pattern più complessi. Uno svantaggio importante di aumentare eccessivamente la dimensione del kernel è che aumenta anche il numero di parametri da apprendere, ma ancor di più aumenta considerevolmente il numero di operazioni da effettuare durante la convoluzione, per tale ragione è consuetudine utilizzare kernel di dimensione 1x1, 3x3 o 5x5, difficilmente si utilizzano kernel di dimensione maggiore.

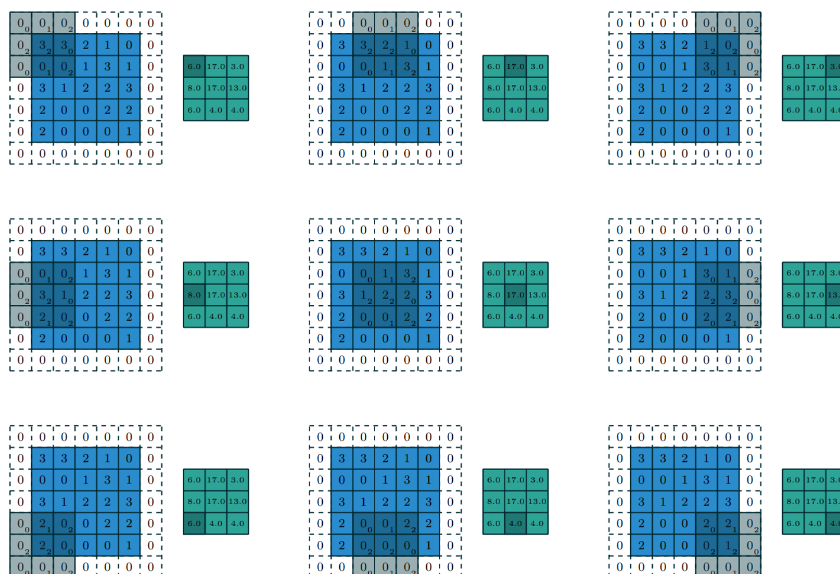


Figura 1.16: Esempio di convoluzione con un kernel 3x3, stride pari a 2, e padding (zero padding) pari a 1.
credits: Dumoulin et al. 2016 [5]

### Numero di filtri

Il numero di filtri è un parametro che viene utilizzato per decidere il numero di filtri da utilizzare per effettuare la convoluzione, tipicamente equivale a decidere quanti kernel diversi si vogliono utilizzare per effettuare la convoluzione, ottenendo altrettanti canali nel tensore di output. Ad esempio se abbiamo un tensore 1x5x5 in input e decidiamo di utilizzare 3 filtri 3x3 con stride 1 e padding 0, otteniamo un tensore 3x3x3 in output, dove ogni canale è il prodotto della convoluzione tra il tensore di input e un kernel diverso.

#### La dimensione di output

Dopo aver visto i principali parametri della convoluzione, vediamo come questi influenzano la dimensione dell'immagine di output. Possiamo calcolare la dimensione dell'immagine di output in base ai seguenti parametri di input:

- $W_{in}, H_{in}$: dimensione del tensore di input.
- $W_k, H_k$: dimensione del kernel.
- $P$: padding.
- $S$: stride.
- $W_{out}, H_{out}$: dimensione del tensore di output.

Ipotizzando lo stride e il padding uguali lungo entrambe le direzioni di convoluzione, le dimensioni di output si calcolano come segue:

$$H_{out} = \frac{H_{in} - H_k + 2P}{S} + 1 \tag{1.20}$$

$$W_{out} = \frac{W_{in} - W_k + 2P}{S} + 1 \tag{1.21}$$

### 1.4.4 Il pooling

Il pooling è un'operazione che viene utilizzata per ridurre la dimensione di un tensore, tipicamente viene utilizzato dopo una o più convoluzioni, in modo da ridurre la quantità di dati da elaborare, rendendo le convoluzioni successive più veloci, e permettendo di lavorare con kernel più piccoli su pattern di più alto livello. Infatti riducendo la dimensione di un tensore diciamo della metà, si potrà lavorare con features più complesse, senza la necessità di utilizzare kernel più grandi dato che saranno le features stesse a venire ridotte in dimensione. Le tipologie principali di pooling sono il max pooling e l'average pooling, entrambe vengono applicate in maniera simile alla convoluzione, con lo scorrimento di una finestra, ma invece di effettuare una combinazione lineare dei valori del tensore con i pesi di un kernel, viene applicata una diversa funzione.

Anche per il pooling vale una regola simile alla convoluzione per il calcolo della dimensione dell'immagine di output, con l'unica differenza che nel pooling solitamente il padding non viene utilizzato, quindi la dimensione dell'immagine di output si calcola come segue:

$$H_{out} = \frac{H_{in} - H_k}{S} + 1 \tag{1.22}$$

$$W_{out} = \frac{W_{in} - W_k}{S} + 1 \tag{1.23}$$

### Max pooling

Con il **max pooling** viene applicata la funzione di massimo, ovvero considerando una data finestra di scorrimento, il tensore di output avrà per ogni valore il valore massimo presente nella corrispondente finestra del tensore di input.

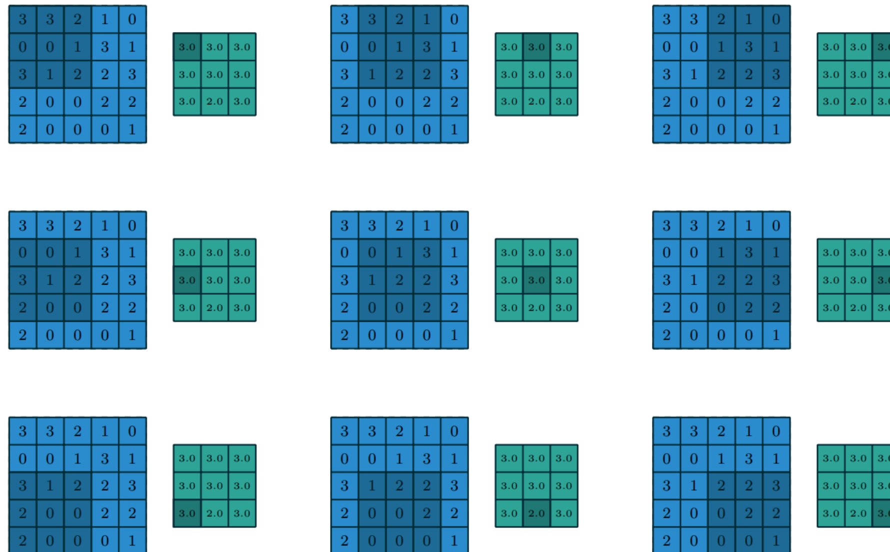


Figura 1.17: Esempio di max pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0.
credits: Dumoulin et al. 2016 [5]

### Average pooling

Con l'**average pooling** viene applicata la funzione di media, ovvero considerando una data finestra di scorrimento, il tensore di output avrà per ogni valore la media dei valori presenti nella corrispondente finestra del tensore di input.

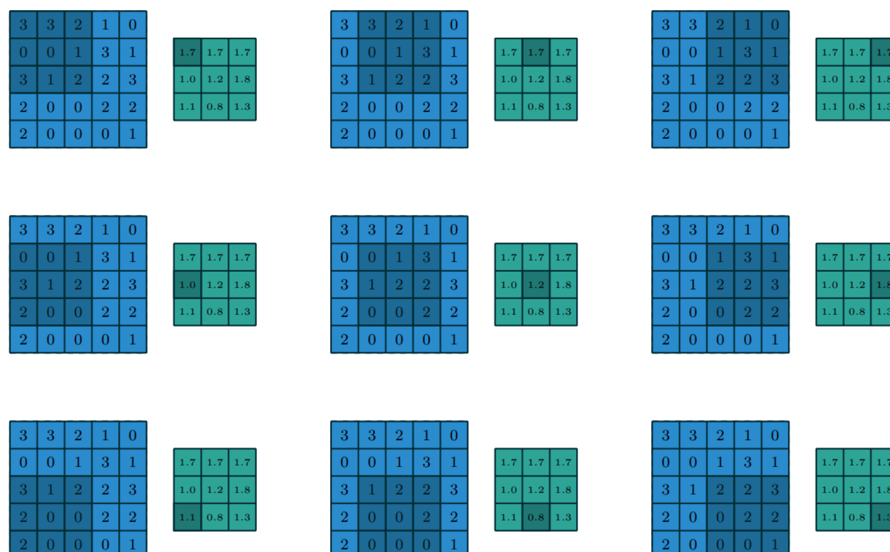


Figura 1.18: Esempio di average pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0.
credits: Dumoulin et al. 2016 [5]

### 1.4.5 La convoluzione multichannel

Diversamente da quanto visto nei precedenti esempi, le operazioni di convoluzione utilizzate nelle CNN sono applicate a tensori 3D, dunque con più canali, perciò si parla di convoluzione 2D su tensori con 3 dimensioni ($W \times H \times C$), dove $W$ e $H$ sono altezza e larghezza del tensore mentre $C$ è il numero di canali. Le immagini tipicamente hanno questa struttura e si ha che i diversi canali enfatizzano diversi aspetti dell'immagine, per tale ragione non si effettua la convoluzione della stessa matrice di pesi su tutti i canali, ma per un dato filtro si ha una matrice di pesi per ogni canale, il risultato delle n convoluzioni viene poi sommato channel-wise per ottenere un canale del tensore di output.

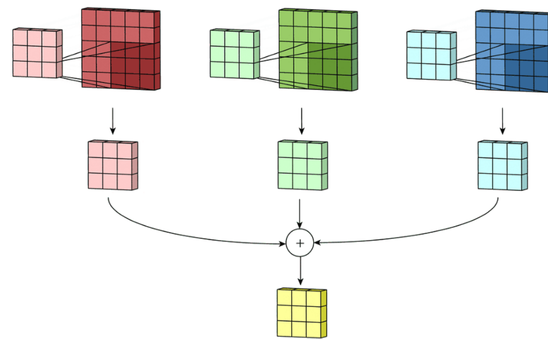


Figura 1.19: Esempio di convoluzione di un'immagine rgb.
credits: Irhum Shafkat [27]

Facciamo però un pò di chiarezza sulla differenza tra kernel e filtro, nella convoluzione 2D, con kernel si intende la matrice di pesi ($W_k \times H_k \times 1$) che viene applicata ad un canale di un tensore, mentre con filtro (Il tensore arancio in 1.20) si intende un tensore composto da $C$ kernel, il quale ha una dimensione $W_k \times H_k \times C$, dove $W_k$ e $H_k$ sono altezza e larghezza del kernel, mentre $C$ è il numero di canali del tensore di input. Per tale ragione in un dato layer convoluzionale che ha in input un tensore con $n$ canali e in output un tensore con $m$ canali si avranno $m$ filtri, composti a loro vota da $n$ kernel, perciò il numero di parametri di tale layer convoluzionale sarà pari a $m \times n \times W_k \times H_k$, dove $W_k$ e $H_k$ sono le dimensioni del kernel.

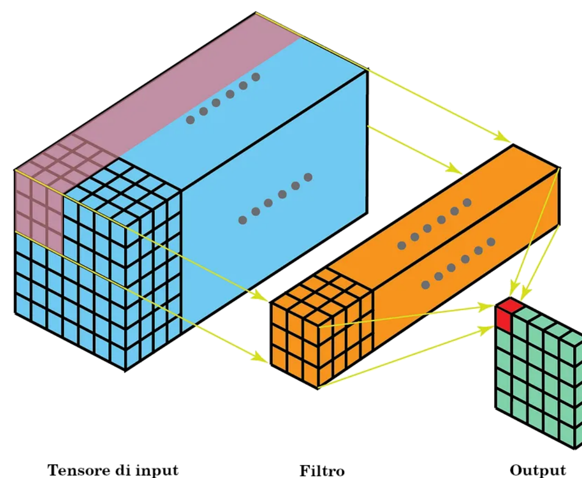



Figura 1.20: Esempio di convoluzione multichannel.
credits: Irhum Shafkat [27]

# Capitolo 2

# Stato dell'arte

In questa sezione verranno mostrati i principali modelli generativi basati su architettura GAN, che sono stati utilizzati negli ultimi anni, partendo dal primo modello proposto da Goodfellow et al. [11] capostipite di questa famiglia di modelli passando per alcune delle principali innovazioni proposte da altri autori, fino ad arrivare al modello LAMA [28], il quale è stato preso come base per questo progetto e riadattato per l'*inpainting* condizionato.

## 2.1 Il primo modello GAN

Nel 2014 Ian Goodfellow et al. [11] hanno proposto il primo modello generativo basato su architettura GAN (*Generative Adversarial Neural Network*). Questa pubblicazione ha segnato un punto di svolta nella ricerca dei modelli di deep learning generativi, i quali fino ad allora erano basati su architetture come gli *autoencoder* o le *deep Boltzmann machines* (DBM) che non sono in grado di generare dati di alta qualità, o se arrivano a buoni risultati tipicamente hanno una distribuzione molto concentrata intorno ai dati del training set, dunque raggiungendo una scarsa generalizzazione.

### 2.1.1 L'adversarial training

Il modello GAN proposto in questa ricerca si componeva di due attori principali, il *discriminatore* **D** e il *generatore* **G**, due modelli neurali MLP, che si affrontano in un gioco a due giocatori a somma zero. Il generatore in questo gioco ha il compito di generare dati che siano in grado di ingannare il discriminatore, che ha il compito opposto di distinguere i dati che provengono dal generatore da quelli che provengono dal training set. Il gioco viene detto a somma zero in quanto il successo di uno dei due attori è sempreassociato al fallimento dell'altro, dunque il gioco non può mai vedere entrambi i giocatori vincitori. I due diventeranno gradualmente sempre più bravi nel loro compito fino al punto in cui la distribuzione dei dati generati dal generatore sarà molto simile a quella dei dati del training set. Una componente importante che differenzia le GAN da altri modelli generativi è la presenza di un input random $\mathbf{z}$, che viene passato al generatore, questo fatto porta il generatore ad una maggiore generalizzazione in quanto questo

cercherà naturalmente una funzione che leghi il suo output ad un input *random*, dunque la sua distribuzione sarà più ampia e non strettamente concentrata intorno ai dati del training set come nel case dei *variational autoencoder* (VAE).

A questo punto, iniziamo ad elencare le componenti che utilizzeremo per descrivere matematicamente come funziona la procedura di addestramento di questa GAN.

- $\mathbf{x}$: esempio proveniente dal training set, $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$.
- $\mathbf{z}$: input casuale, utilizzato per determinare una mappatura nello spazio dei dati, $\mathbf{z} \in \mathbb{R}^{\mathbf{m}}$.
- $\mathbf{y}$: uscita del discriminatore, $\mathbf{y} \in [\mathbf{0}, \mathbf{1}]$, esprime la probabilità che x sia reale.
- $\hat{\mathbf{y}}$: uscita desiderata, $\hat{\mathbf{y}} \in \{\mathbf{0}, \mathbf{1}\}$, può assumere solo due valori, vero o falso.
- $\mathbf{p_z}$: distribuzione dei vettori casuali passati al generatore.
- $\mathbf{p_g}$: distribuzione dei dati generati dal generatore.
- $\mathbf{p_{data}}$: distribuzione dei dati del training set.
- $\theta_G$: parametri del generatore.
- $\theta_D$: parametri del discriminatore.
- $\mathbf{G}(\theta_\mathbf{G}, \mathbf{z})$: Funzione generatore che mappa lo spazio random $z$ nello spazio dei dati, attraverso i parametri $\theta_G$. definibile come una funzione $\mathbf{f} : \mathbb{R}^{\mathbf{m}} \to \mathbb{R}^{\mathbf{n}}$.
- $\mathbf{D}(\theta_\mathbf{D}, \mathbf{x})$: Funzione discriminatore che mappa lo spazio dei dati in un uno spazio $\{\mathbf{0}, \mathbf{1}\}$, attraverso i parametri $\theta_D$. definibile come una funzione $\mathbf{f} : \mathbb{R}^{\mathbf{n}} \to \{\mathbf{0}, \mathbf{1}\}$. Tale spazio identifica la provenienza di un esempio x come segue:
  - se $\mathbf{D}(\theta_\mathbf{D}, \mathbf{x}) = 1 \to \mathbf{x} \in \mathbf{p_{data}}$.
  - se $\mathbf{D}(\theta_\mathbf{D}, \mathbf{x}) = 0 \to \mathbf{x} \in \mathbf{G}(\theta_\mathbf{G}, \mathbf{z}), z \sim p_z$.

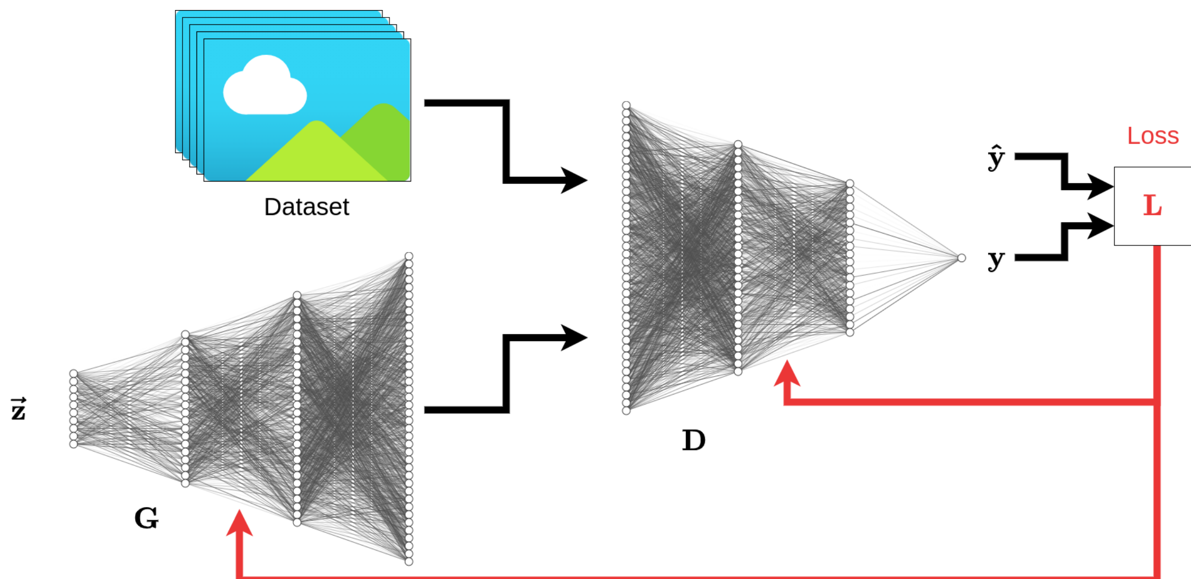



Figura 2.1: Rappresentazione grafica della training pipeline di un modello GAN.

### 2.1.2 La loss function

L'*adversarial training* si basa su un concetto innovativo, invece di cercare di definire matematicamente una *loss function* in grado di guidare il generatore durante l'addestramento, cerca di apprenderne una, nello specifico il discriminatore rappresenta la *loss function* che viene addestrata. Per dare un'idea di che tipo di miglioramento ha introdotto questo approccio facciamo un esempio utilizzando una *loss function* molto utilizzata per addestrare modelli generativi prima dell'introduzione dei modelli GAN, la *Mean Squared Error* (MSE). Questa *loss function*, specialmente per generatori di immagini, portava a dei risultati sfocati e poco realistici, in quanto la MSE non è in grado di catturare la struttura dei dati, ma solo di portare il generatore verso un punto medio tra i dati del training set, che non necessariamente fa parte della distribuzione, tale problema è mostrato graficamente nella seguente figura (Figura: 2.2).

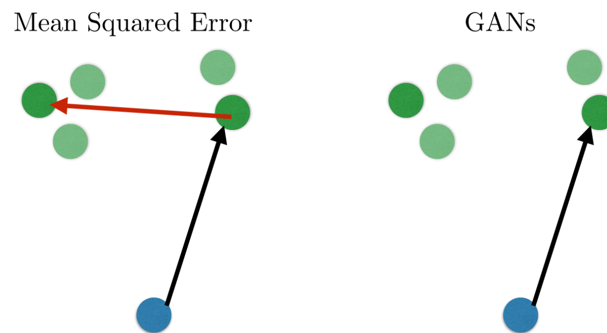


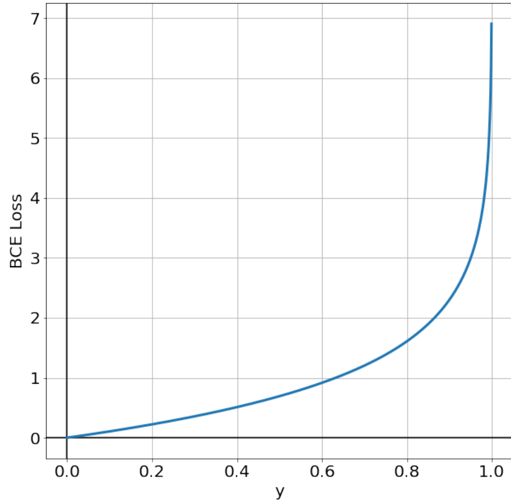

Figura 2.2: Confronto tra MSE e Adversarial training.
credits: Goodfellow [10]

Nella figura possiamo vedere in verde un ipotetico insieme di dati reali e in blu i dati generati dal generatore. Nel caso del MSE vediamo come il gradiente della loss spinge il generatore verso la zona centrale in quanto punta alla media dei dati reali, anche se quella zona non fa parte della distribuzione, mentre nel caso dell'*adversarial training* il gradiente punta verso i dati, in quanto questa loss è in grado di fare delle considerazioni locali sulla distribuzione dei dati, evitando di cadere in minimi lontani dalla distribuzione reale.

Per addestrare il discriminatore e apprendere così una *loss function* aderente ai dati, si è utilizzata come la *Binary Cross Entropy* (BCE), utilizzata tipicamente per la classificazione binaria, la quale risulta logicamente adeguata per valutare l'output del discriminatore.
La BCE in generale è definita come segue:

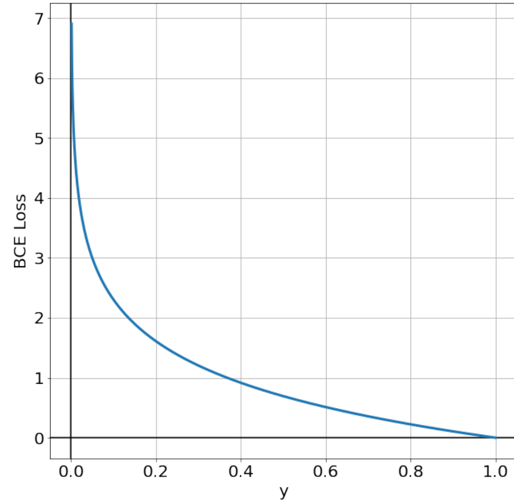
$$\mathbf{BCE}(\mathbf{y}, \hat{\mathbf{y}}) = -\hat{\mathbf{y}} \cdot \log(\mathbf{y}) - (\mathbf{1} - \hat{\mathbf{y}}) \cdot \log(\mathbf{1} - \mathbf{y}) \tag{2.1}$$

Presenta 2 componenti principali, $-\hat{\mathbf{y}} \cdot \log(\mathbf{y})$ che si attiva quando il risultato atteso è $\mathbf{y} = \mathbf{1}$, mentre l'altra componente si azzera, e $(\mathbf{1} - \hat{\mathbf{y}}) \cdot \log(\mathbf{1} - \mathbf{y})$ che si attiva quando il risultato atteso è $\mathbf{y} = \mathbf{0}$, con l'altra componente a 0.




(a) Binary Cross Entropy ($\hat{y} = 0$)

$-\log(\mathbf{1} - \mathbf{y})$




(b) Binary Cross Entropy ($\hat{y} = 1$)

$-\log(\mathbf{y})$

Nel caso dell'addestramento di un modello GAN, possiamo riscrivere la BCE, attuando alcune semplificazioni, e sostituendo **y** con il valore corrispondente, nel caso in cui la **y** deriva da un sample reale e quando deriva da un sample generato:

$$\hat{\mathbf{y}} = \mathbf{1} \rightarrow \mathbf{y} = \mathbf{D}(\theta_{\mathbf{D}}, \mathbf{x}) \tag{2.2}$$

$$\hat{\mathbf{y}} = \mathbf{0} \rightarrow \mathbf{y} = \mathbf{D}(\theta_{\mathbf{D}}, \mathbf{G}(\theta_{\mathbf{G}}, \mathbf{z})) \tag{2.3}$$

Trasformando la BCE loss in un gioco di minimizzazione e massimizzazione contrapposta tra i due attori, ottenendo la seguente funzione:

$$\mathbf{min_G max_D V(D, G)} = \mathbb{E}_{\mathbf{x} \sim \mathbf{p_{data}}}[\mathbf{log(D(x))}] + \mathbb{E}_{\mathbf{z} \sim \mathbf{p_z}}[\mathbf{log(1 - D(G(z)))}] \tag{2.4}$$

In generale possiamo riscrivere le due componenti della funzione appena descritta in maniera estesa come segue:

$$\mathbb{E}_{x \sim p_{data}}[log(D(x))] = \sum_{i=1}^{n} log(D(x_i)), x_i \in \mathbb{D}t \tag{2.5}$$

Dove $\mathbb{D}t = \{x_1, x_2, ..., x_n\}$ è l'insieme degli esempi del training set. Mentre la seconda componente è definita come segue:

$$\mathbb{E}_{z \sim p_z}[log(1 - D(G(z)))] = \sum_{i=1}^{n} log(1 - D(G(z_i))), z_i \sim p_z \tag{2.6}$$

L'applicazione di questa funzione però necessita di qualche accorgimento, nella pratica infatti

il modello viene addestrato in due fasi distinte e periodiche, in una viene addestrato il discriminatore, e nell'altra il generatore, per un numero di step predefiniti. Tale dinamica di addestramento ha lo scopo di mantenere le uscite del discriminatore per i dati generati e reali sufficientemente vicine da non causare problemi di saturazione. Altrimenti la saturazione comporta un azzeramento del gradiente e di conseguenza un arresto dell'apprendimento.

### 2.1.3 La convergenza del generatore

Per visualizzare meglio le dinamiche dell'addestramento che portano alla convergenza del generatore alla distribuzione del training set possiamo utilizzare un grafico, ipotizzando di visualizzare i dati del training set su una sola dimensione, e di visualizzare sull'asse $y$ la densità di probabilità dei dati e il valore dell'uscita del discriminatore, al variare di $\mathbf{x}$, ossia della variabile di input del discriminatore, in questo caso unidimensionale ma in generale potrebbe rappresentare immagini, audio video e così via.

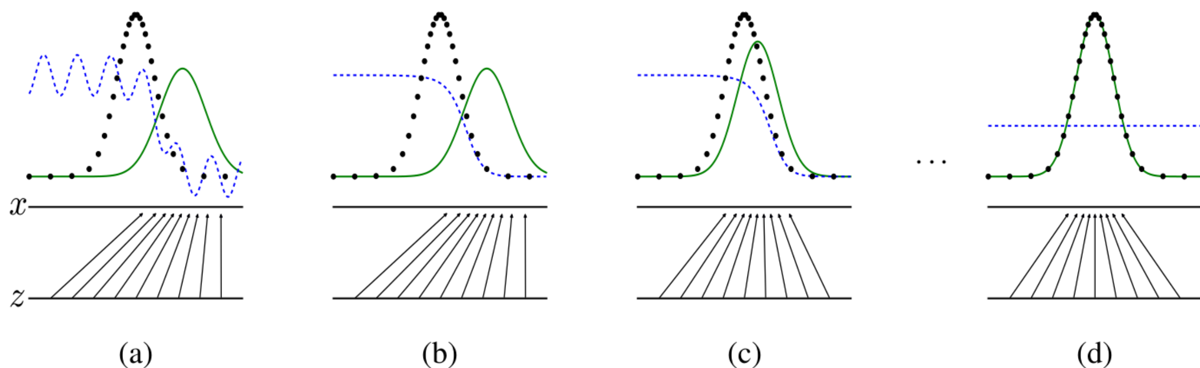



Figura 2.4: Visualizzazione schematica della convergenza della distribuzione dei dati generati verso quelli reali, e dell'uscita del discriminatore al variare di $\mathbf{x}$.
Si noti che la linea nera tratteggiata rappresenta la distribuzione dei dati reali $p_{data}$, mentre la linea verde continua rappresenta la distribuzione dei dati generati $p_{model}$ e in fine la linea blu tratteggiata rappresenta l'uscita del discriminatore $D(x)$, al variare di $\mathbf{x}$. I 2 assi in fondo sono rispettivamente, l'asse $z$ i valori casuali di input del generatore con distribuzione fissa, e l'asse $x$, i valori di input del discriminatore, appartenenti alla distribuzione $\mathbf{p_{data}}$ o $\mathbf{p_g}$.
credits: Goodfellow et al. [11]

Nella figura 2.4 possiamo vedere diverse fasi dell'addestramento, considerando 4 istanti successivi (a, b, c, d) abbiamo che (a) presenta una situazione di apprendimento intermedio in cui il discriminatore comincia ad essere in grado di distinguere con qualche difficoltà i dati reali da quelli generati e il generatore produce dati piuttosto vicini a quelli reali, in (b) il discriminatore è in grado di distinguere con maggiore precisione i dati reali da quelli generati, fornendo uno stimolo migliore al generatore, che in (c) è in grado di mappare la distribuzione $\mathbf{p_z}$ ad una distribuzione $\mathbf{p_g}$ che tende sempre meglio a quella dei dati reali, infine in (d) possiamo osservare il caso del raggiungimento dell'ottimalità del generatore e dunque la fine dell'addestramento. In questo caso il discriminatore non è più in grado di distinguere i dati reali da quelli generati, in quanto la distribuzione $\mathbf{p_g}$ è sovrapposta a quella dei dati reali $\mathbf{p_{data}}$, e si ottiene un'uscita del generatore uguale a 0.5 per ogni valore di input. La condizione di ottimalità può essere

dimostrata, infatti possiamo definire il discriminatore come segue:

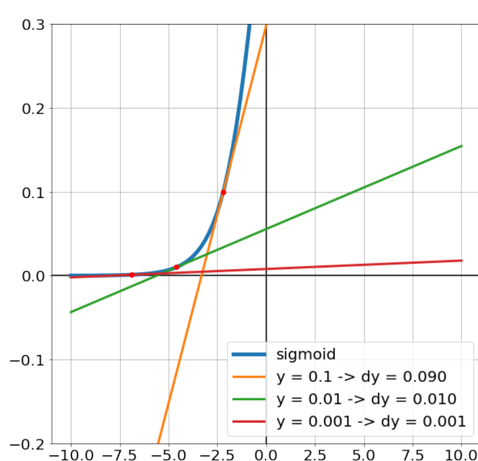
$$\mathbf{D}(\mathbf{x}) = \frac{\mathbf{p_{data}}(\mathbf{x})}{\mathbf{p_{data}}(\mathbf{x}) + \mathbf{p_g}(\mathbf{x})} \tag{2.7}$$

Sapendo che nella condizione ottimale le distribuzioni sono coincidenti $\mathbf{p_{data}}(\mathbf{x}) = \mathbf{p_g}(\mathbf{x})$, allora considerando $\mathbf{D}^*$ il discriminatore ottimo per un dato $\mathbf{G}$, possiamo scrivere:

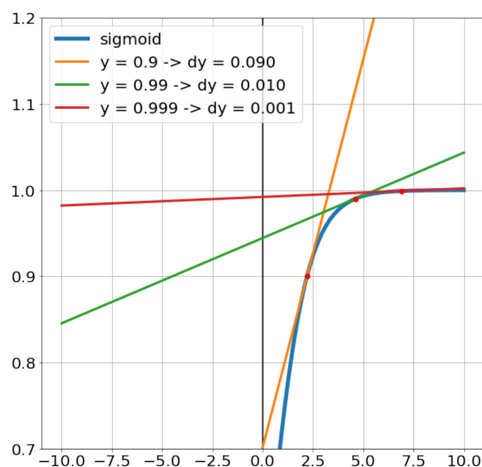
$$\mathbf{D}^*(\mathbf{x}) = \frac{\mathbf{p_{data}}(\mathbf{x})}{\mathbf{p_{data}}(\mathbf{x}) + \mathbf{p_g}(\mathbf{x})} = \frac{\mathbf{p_{data}}(\mathbf{x})}{\mathbf{2p_{data}}(\mathbf{x})} = \frac{\mathbf{1}}{\mathbf{2}} \tag{2.8}$$

### 2.1.4 Il gradient vanishing

Il gradient vanishing è un problema che si verifica nel momento in cui il discriminatore riesce a distinguere con troppa facilità i dati reali da quelli generati, consideriamo infatti che questa è caratterizzata da un singolo elemento dotato di funzione di trasferimento sigmoide, la quale nel momento in cui restituisce un valore troppo vicino a 0 o 1, ha il gradiente tendente a zero, e di conseguenza dal momento che l'addestramento viene effettuato tramite backpropagation, per quanto visto nel capito precedente anche i gradienti dei layer precedenti tenderanno a zero, e l'addestramento si arresterà. Vediamo di eseguito la funzione di trasferimento sigmoide e il suo gradiente nel momento in cui tende a saturare:



(a) Saturazione per $\mathbf{y} \to \mathbf{0}$



(b) Saturazione per $\mathbf{y} \to \mathbf{1}$

### 2.1.5 Il mode collapse

Un'altro problema che affligge i modelli gan è quello del mode collapse, che si verifica quando $\mathbf{G}$ apprende una distribuzione che concide con un sottoinsieme di $\mathbf{p_{data}}$. Tale condizione consente comunque di raggiungere la condizione di ottimalità sopra descritta con $p_g \subset p_{data}$, ottenendo comunque $\mathbf{D}^*(\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{2}}$. Per fare un esempio pratico possiamo considerare ad esempio il dataset Minst che contiene immagini di cifre scritte a mano, se il generatore impara a generare soltanto cifre dallo 0 al 5 e non quelle dal 6 al 9, siamo in una condizione di mode collapse, in quanto parte della distribuzione dei dati non è stata appresa, ma il sistema può comunque raggiungere la

condizione di ottimalità. Nella seguente immagine si ha nella prima riga un esempio di corretto apprendimento mentre nella seconda riga si ha un esempio di mode collapse.

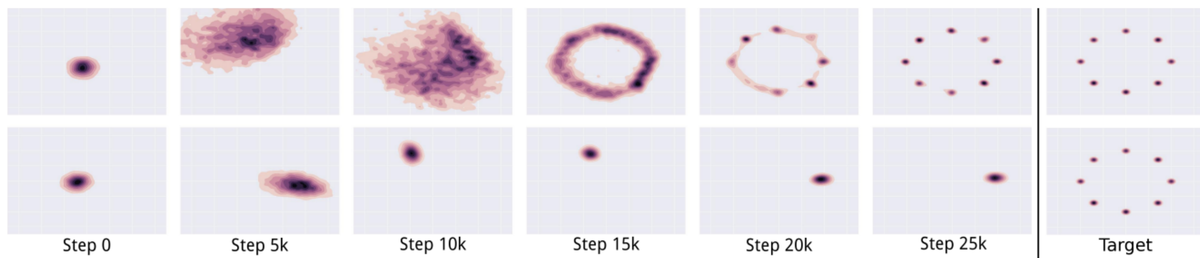



Figura 2.6: Esempio di corretto apprendimento (prima riga) e di mode collapse (seconda riga). credits: Luke Metz et al. [22]

### 2.1.6 Alcuni risultati

Vediamo in questa sezione alcuni risultati relativi al modello gan presentato da Goodfellow, relativi a modelli addestrati con dataset di facce umane in bassa risoluzione e il dataset Minst. In Questi esempi è possibile vedere sulla destra con contorno giallo alcuni esempi di dati provenienti dal dataset di addestramento, mentre sulla sinistra ci sono gli esempi generati dal modello.

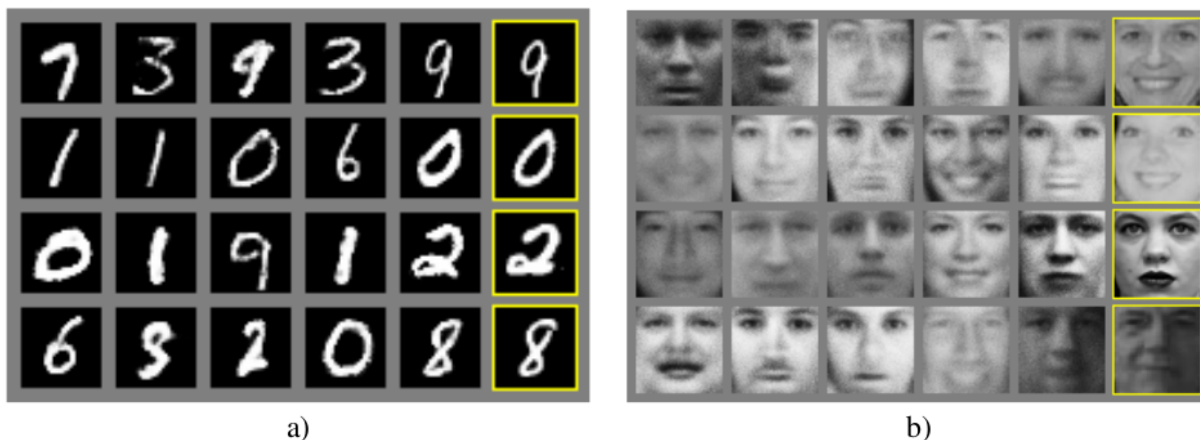



Figura 2.7: Alcuni risultati del modello addestrato in questa ricerca, relativi a un dataset di facce umane in bassa risoluzione (b) e il dataset Minst (a).
credits: Goodfellow et al. [11]

## 2.2 DCGAN

DCGAN è stata la prima pubblicazione, opera di Alec Radford et. al [24], che ha dimostrato l'applicabilità dell'adversarial training su modelli convoluzionali, ottenendo buoni risultati, aumentando la dimensione delle immagini rispetto a lavori precedenti e mantenendo una buona efficienza computazionale. Inoltre in questo articolo sono state mostrate per la prima volta le proprietà aritmetiche vettoriali dello spazio latente appreso dal generatore durante l'addestramento.

### 2.2.1 L'architettura del modello

In realtà questo non è stato il primo tentativo di applicare l'adversarial training su modelli convoluzionali, ma il primo ad avere successo. Altri ci hanno provato prima ma con scarsi risultati, principalmente a causa dell'instabilità del training, che in questo particolare lavoro è stata risolta con l'uso di alcuni interessanti accorgimenti.

Rispetto alle implementazioni classiche di convolutional neural network, una scelta interessante è stata quella di rimuovere completamente i layer fully connected, eccetto per l'uscita del discriminatore che è un singolo neurone con funzione di attivazione tanh. Inoltre sono stati rimossi completamente i layer di pooling, sostituiti da semplici layer di convoluzione con stride 2, in modo da dare al modello la possibilità di apprendere in autonomia come applicare il downsampling nel discriminatore. Nel generatore invece sono stati usati layer di convoluzione trasposta con stride 2, per ottenere l'upsampling. Altre note interessanti riguardano l'uso della funzione di attivazione LeakyReLU per il discriminatore e della ReLU per il generatore, e l'uso della batch normalization per entrambi i modelli, che ha permesso di ottenere una maggiore stabilità del training.

Di seguito vediamo una delle architetture utilizzate in questo articolo per il generatore:

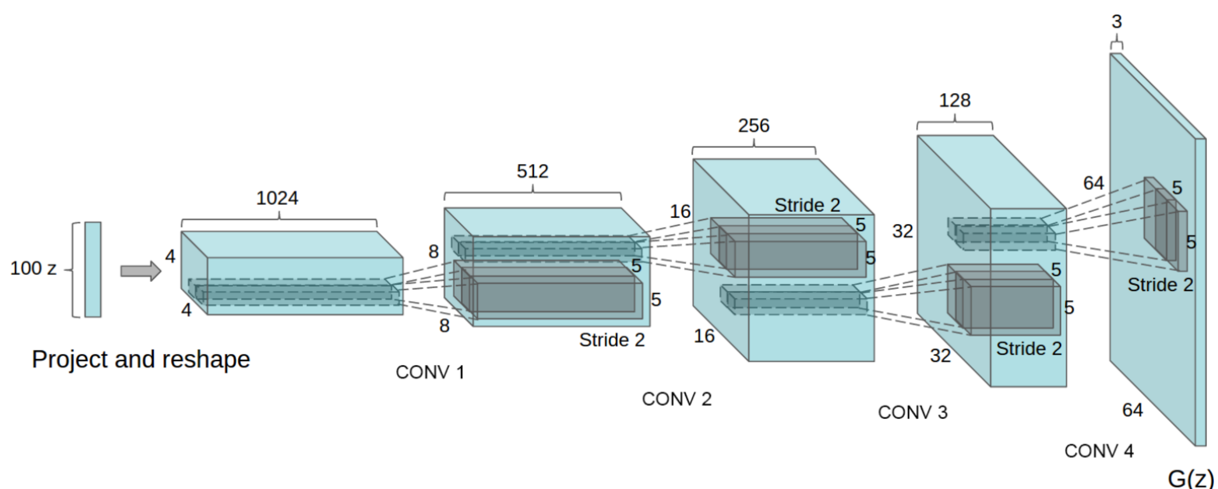



Figura 2.8: Architettura del generatore di DCGAN.
credits: Alec Radford et al. [24]

### 2.2.2 L'algebra vettoriale nello spazio latente Z

Un'altra proprietà interessante messa in luce da questo articolo sui modelli GAN è il fatto che durante l'addestramento, benché l'input $\mathbf{z} \sim \mathbf{p_z}$ sia un vettore casuale proveniente da una distribuzione tipicamente gaussiana, il generatore mostra delle proprietà di auto organizzazione, associando arbitrariamente diverse zone dello spazio latente a diverse caratteristiche appartenenti alla distribuzione dei dati reali. Questo spazio autorganizzato mostra addirittura delle proprietà aritmetiche vettoriali, in quanto sembra possibile effettuare operazioni di somma e sottrazione tra i vettori in Z, ed ottenere dei risultati coerenti in termini di immagini. Le stesse semplici operazioni effettuate a livello di immagine non danno risultati paragonabili, per tale ragione questo comportamento è indice del fatto che il modello apprende una rappresentazione dei dati nello spazio latente in maniera profonda e relativa alle caratteristiche che essi presentano, in maniera del tutto non supervisionata.

Vediamo un esempio tratto dallo stesso articolo, creato attraverso un modello addestrato su un dataset di facce umane. Nell'esempio vengono presi 3 vettori per 3 distinte aree di Z: "donna sorridente", "donna seria", "uomo serio" e viene fatta la media di questi vettori, ottenendo ancora un vettore che possiede le stesse caratteristiche, ciò indica che queste tre aree sembrano avere una sorta di convessità, i vettori mediati vengono poi utilizzati per estrarre la caratteristica "sorridente" e aggiungerla al vettore appartenente all'area "uomo serio", ottenendo un nuovo vettore che rappresenta un uomo sorridente. Ciò significa che nello spazio Z è possibile isolare delle caratteristiche e combinarle tra loro per ottenere nuove immagini, lo svantaggio purtroppo è che lo spazio Z è caratterizzato da una dimensionalità molto elevata, e inoltre queste zone associate a caratteristiche specifiche presentano una elevata non linearità, per cui in generale è difficile effettuare operazioni di questo tipo con un buon controllo del risultato.

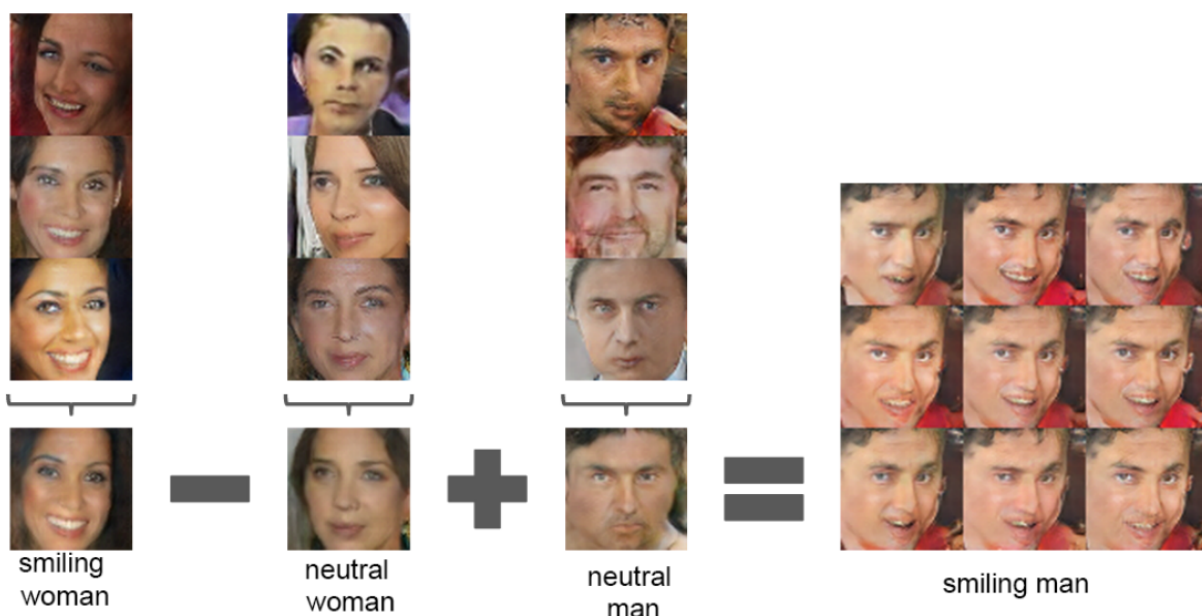



Figura 2.9: Esempio di algebra vettoriale nello spazio latente Z.
credits: Alec Radford et al. [24]

## 2.3 Wasserstein GAN

Uno dei problemi più seri relativi all'addestramento di un modello neurale attraverso l'*adversarial training*, è la dissolvenza del gradiente durante il training, che porta ad un situazione di stallo in cui il generatore non è più in grado di apprendere. Tale condizione viene raggiunta nel momento in cui il discriminatore diventa troppo efficacie nel distinguere gli esempi appartenenti al dataset reale da quelli generati dal generatore, in quel caso come già visto precedentemente nella sezione 2.1.4, il gradiente dell'errore si annulla a causa della saturazione della funzione di attivazione dell'uscita del discriminatore. Vediamo di seguito un esempio che mette in luce questo problema ipotizzando di avere 2 distribuzioni di dati unidimensionali, una reale $P_{data}$ e una generata dal generatore $P_G$, tali distribuzioni sono affiancate dall'uscita del discriminatore $D(x)$ al variare di $x$.

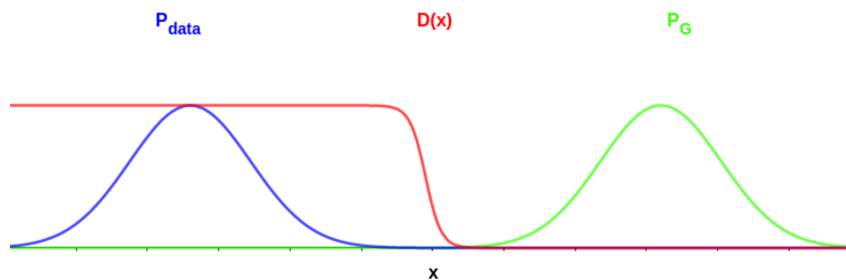



Figura 2.10: Esempio di *gradient vanishing*, causato da un discriminatore addestrato fino all'ottimo per un generatore non ottimo.

Una soluzione a questo problema è stata proposta da Martin Arjovsky et al. [1] nel 2017, i quali hanno introdotto una nuova tipologia di loss function che permette di evitare il problema del *gradient vanishing* e di ottenere una migliore stabilità del training, questa rivisitazione del modello GAN è stata chiamata *Wasserstein GAN* (WGAN). Per comprendere l'innovazione introdotta da questo lavoro dobbiamo dare uno sguardo alla funzione divergenza alla base della funzione di loss del precedente modello GAN, ovvero la *Jensen-Shannon divergence* (JSD), che è definita come segue:

$$JSD(P_{data}, P_G) = \frac{1}{2}KL(P_{data}||\frac{P_{data} + P_G}{2}) + \frac{1}{2}KL(P_G||\frac{P_{data} + P_G}{2}) \tag{2.9}$$

Questa metrica permette di misurare la distanza tra due distribuzioni di probabilità, e viene utilizzata in quanto presenta due importanti proprietà, ovvero è simmetrica ( $JSD(P_{data}, P_G) = JSD(P_G, P_{data})$ ) ed è sempre definita. La $JSD$ è stata introdotta per risolvere le problematiche della $KL$ *Kullback-Leibler divergence*, che non è simmetrica, ovvero $(KL(P_{data}||P_G) \neq KL(P_G||P_{data}))$, e non è sempre definita. In ogni caso quest'ultima è la formulazione matematica alla base della comunemente utilizzata *cross-entropy loss.* vediamo di seguito la definizione della $KL$:

$$KL(P_{data}||P_G) = \sum_x P_{data}(x) log \frac{P_{data}(x)}{P_G(x)} \tag{2.10}$$

Nonostante la $JSD$ abbia dimostrato la sua efficacia con la precedente formulazione del modello GAN, essa risulta poco sensibile alla distanza tra due distribuzioni di probabilità, in particolare quando queste sono molto diverse tra loro, come è possibile vedere graficamente in figura 2.10.

### 2.3.1 Earth-Mover distance

Per risolvere questo problema gli autori hanno formulato una metrica alternativa, chiamata *Earth-Mover distance* (EMD), che concettualmente misura quanto deve essere spostata una distribuzione per farla coincidere con l'altra. Concettualmente questa metrica è molto più sensibile di tutte le altre divergenze come la $KL$ e la $JSD$ e altre, in quanto tiene conto della distanza in maniera lineare. Vediamo di seguito la definizione matematica della $EMD$ per la quale utilizzeremo la notazione $W(P_{data}, P_G)$:

$$W(P_{data}, P_G) = \inf_{\gamma \in \prod(P_{data}, P_G)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||] \tag{2.11}$$

Analizzando le componenti di questa formula possiamo notare che il termine $\prod(P_{data}, P_G)$ rappresenta l'insieme di tutte le possibili distribuzioni di probabilità $\gamma$ che hanno come marginali le distribuzioni $P_{data}$ e $P_G$, ovvero $\gamma$ è una distribuzione di probabilità congiunta. Per tale ragione il fatto che $W(P_{data}, P_G)$ sia definita come l'estremo inferiore di tutte le possibili distribuzioni congiunte $\gamma$ che minimizzano la distanza tra le due distribuzioni $P_{data}$ e $P_G$, può essere interpretato come il percorso minimo che deve essere fatto da una distribuzione per trasformarsi nell'altra.

Il problema di questa formulazione è che non è possibile calcolare direttamente la $EMD$, in quanto non è possibile calcolare la distribuzione congiunta $\gamma$ che minimizza la distanza, per tale ragione gli autori hanno proposto una formulazione alternativa che permette di calcolare la $EMD$ in maniera approssimata, questa formulazione è la seguente:

$$W(P_{data}, P_G) = \sup_{||f||_L \leq 1} \mathbb{E}_{x\sim P_{data}}[f(x)] - \mathbb{E}_{x\sim P_G}[f(x)] \tag{2.12}$$

L'approssimazione proposta consiste nel calcolare la differenza tra il valore atteso di una funzione $f$ calcolata sui dati reali e il valore atteso della stessa funzione calcolata sui dati generati dal generatore, dove $f$ è una funzione arbitraria che rispetta la *Lipschitz continuity* con $K = 1$, ovvero:

$$||f||_{L\leq K} \implies \frac{|f(x) - f(y)|}{|x - y|} \leq K, \forall x \neq y \tag{2.13}$$

In altre parole una funzione che rispetta la *K-Lipschitz continuity* è una funzione che non può avere pendenza (o la derivata) maggiore di $K$ o minore di $-K$, vediamo di seguito nella figura 2.11 e 2.12 due funzioni, una che rispetta la *1-Lipschitz continuity* e una no:

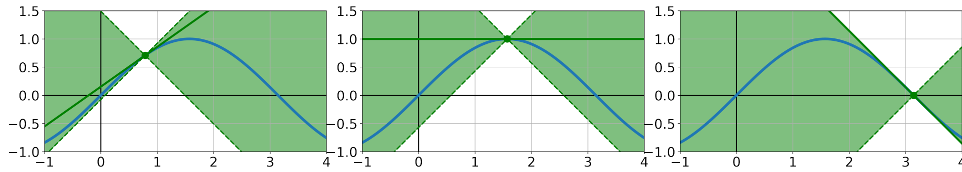


Figura 2.11: In questa immagine è possibile vedere la derivata della funzione seno, nei punti $\pi/4$, $\pi/2$ e $\pi$, ed è possibile vedere come tale funzione rispetta la *1-Lipschitz continuity*, essendo la sua derivata all'interno dell'intervallo ammesso in ogni punto.

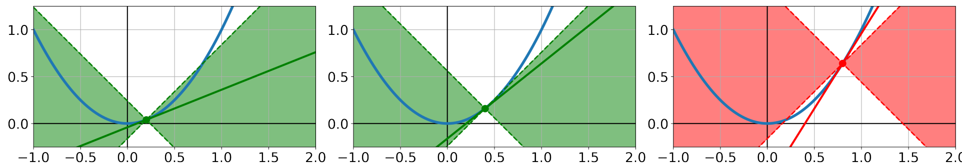


Figura 2.12: In questa immagine è possibile vedere la derivata della funzione $x^2$, nei punti 0.2, 0.4 e 0.8, in questo caso la funzione non rispetta la *1-Lipschitz continuity*, in quanto la sua derivata cresce rapidamente oltre il limite consentito dopo x=0.4.

La funzione $f$ considerata nella equazione 2.12 è detta *critico*, e prende il posto del discriminatore. Concettualmente il *critico* e il discriminatore sono uguali eccetto che per l'uscita, infatti il *critico* non ha una funzione di trasferimento che vincola l'uscita tra 0 e 1, ma può assumere qualsiasi valore reale. Questa peculiarità gli consente di esprimere la distanza tra due distribuzioni in maniera molto più precisa rispetto al discriminatore, che a causa dell'equazione 2.9 alla base del suo funzionamento non è in grado di esprimere la distanza tra due distribuzioni in maniera efficacie. Vediamo di seguito l'immagine 2.13 che mostra come il critico sia in grado di mappare molto più efficacemente la distanza tra due distribuzioni, restituendo dei gradienti utili per il generatore anche quando il critico è addestrato all'ottimo, per il dato generatore.

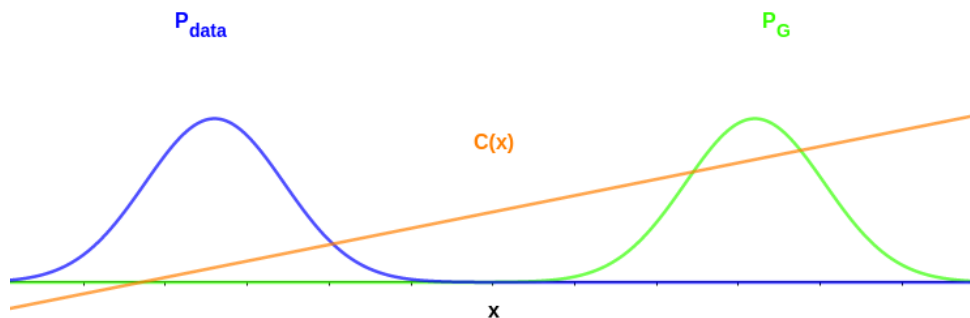



Figura 2.13: In questa immagine è possibile vedere come il critico $C(x)$ sia in grado di mappare efficacemente la distanza tra due distribuzioni $P_{data}$ e $P_G$, restituendo dei gradienti utili per il generatore anche quando il critico è addestrato all'ottimo per il dato generatore.

Per quanto riguarda la *1-Lipschitz continuity*, fare in modo che un modello neurale la rispetti non è un problema banale, nel quale gli stessi autori hanno trovato delle difficoltà, proponendo come soluzione il *weight clipping*, ovvero limitare i pesi del critico ad un intervallo di valori, nello specifico nella paper originale è stato proposto l'intervallo $[-0.01, 0.01]$. Questa soluzione però può portare a problematiche di *vanishing gradient*, un'altra volta, gli autori in ogni caso hanno lasciato aperta la questione incoraggiando la ricerca di soluzioni alternative, che potessero

risolvere il problema in maniera più efficacie e senza effetti collaterali. Le soluzioni che sono state proposte successivamente per risolvere il problema della *1-Lipschitz continuity* sono state principalmente due:

- **Gradient Penalty**: Questa soluzione consiste nel penalizzare il gradiente del critico quando questo assume valori che non rispettano la *1-Lipschitz continuity*.
- **Spectral Normalization**: Questa soluzione consiste nel normalizzare gli autovalori della matrice dei pesi del critico.

### 2.3.2 Alcuni risultati

Vediamo di seguito alcuni risultati, che evidenziano l'andamento dell'addestramento di due modelli, una DCGAN e una MLP a 4 strati, con delle immagini generate durante l'addestramento, e il valore della *loss function*, nell'immagine 2.14 possiamo vedere la *Wasserstein distance* e nell'immagine 2.15 il caso di addestramento con la *JSD*. Tale esempio mette in risalto il fatto che la loss proposta dagli autori dimostra una notevole correlazione con la qualità delle immagini generate, al contrario l'approccio basato su *Jensen-Shannon divergence* non è in grado di fornire un indicatore di qualità delle immagini generate, in quanto la metrica rimane stabile durante l'addestramento o addirittura cresce mediamente, con il miglioramento della qualità delle immagini generate.

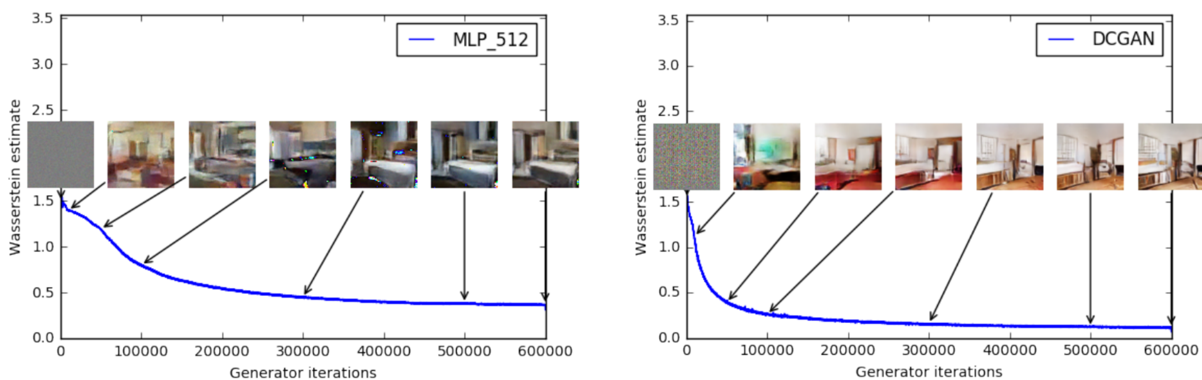



Figura 2.14: Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Wasserstein distance*. credits: Martin Arjovsky et al. [1]

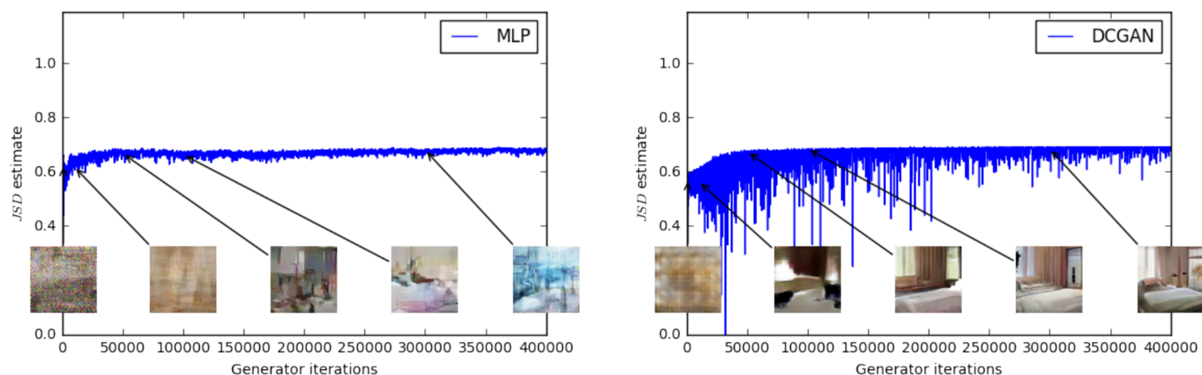



Figura 2.15: Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Jensen-Shannon divergence*. credits: Martin Arjovsky et al. [1]

## 2.4 Pix2Pix

Un'interessante lavoro presentato poco tempo dopo WGAN è stato Pix2Pix da parte di Phillip Isola et al. [15] nel 2018, i quali hanno proposto un'interessante architettura per la generazione di immagini a partire da un'immagine di input. Questa pubblicazione non è stata la prima a presentare un modello che effettuasse la conversione da un'immagine ad un'altra, ma è stata la prima a proporne una variante che sfruttasse l'adversarial training e potesse essere applicata a diversi problemi senza necessità di dover riprogettare l'architettura del modello o la *loss function*.

Molte altre pubblicazioni infatti prima di questa hanno ottenuto risultati molto interessanti per quanto riguarda task che effettuavano la conversione da un'immagine ad un'altra, come ad esempio: la *colorization*, la *super-resolution*, lo *style transfer*, il *denoising*, il *future frame prediction* e molti altri, tutti però hanno in comune il fatto di essere architetture o framework specializzati, realizzati appositamente per quel task, infatti la realizzazione di tali progetti ha richiesto competenze specifiche e un accurato studio del problema. Pix2Pix invece si propone come un'architettura che può essere applicata a diversi task, senza necessità di dover riprogettare l'architettura del modello o la *loss function*, ma semplicemente cambiando il dataset di addestramento, in quanto la *loss function* viene appresa in maniera automatica dal discriminatore durante l'addestramento e la struttura del modello si presta bene a numerosi task.

### 2.4.1 La loss function

La *loss function* proposta da Pix2Pix è una variante della *loss function* proposta nella pubblicazione originale della GAN [11], infatti nonostante questa paper risulta essere pubblicata quasi un'anno dopo la pubblicazione di WGAN, il metodo non si era ancora imposto come standard per l'addestramento di modelli GAN. Sulla base di precedenti lavori infatti in Pix2Pix è stato utilizzato un discriminatore condizionato, ovvero un discriminatore che riceve in input, oltre all'uscita del generatore o l'immagine proveniente dalla distribuzione obbiettivo, anche l'immagine di input, in modo tale che il discriminatore possa valutare la qualità dell'immagine generata anche in base a quest'ultima. Di seguito vediamo l'equazione 2.14 che rappresenta la *loss function* proposta per Pix2Pix detta *Conditional Adversarial Loss*:

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x \sim P_X, y \sim P_Y}[\log D(x, y)] + \mathbb{E}_{x \sim P_X, z \sim P_Z}[\log(1 - D(x, G(x, z)))] \tag{2.14}$$

Si noti che in questa equazione, rispetto al caso classico precedentemente mostrato, spiccano due componenti distinte:

- $\mathbf{x}$ è l'immagine di input
- $\mathbf{P_X}$ è la distribuzione delle immagini di input
- $\mathbf{y}$ è l'immagine di output
- $\mathbf{P_Y}$ è la distribuzione delle immagini di output

Di seguito è mostrato un esempio di training di Pix2Pix per il task *edges to photo*, il quale propone una rappresentazione grafica dell'equazione appena mostrata:

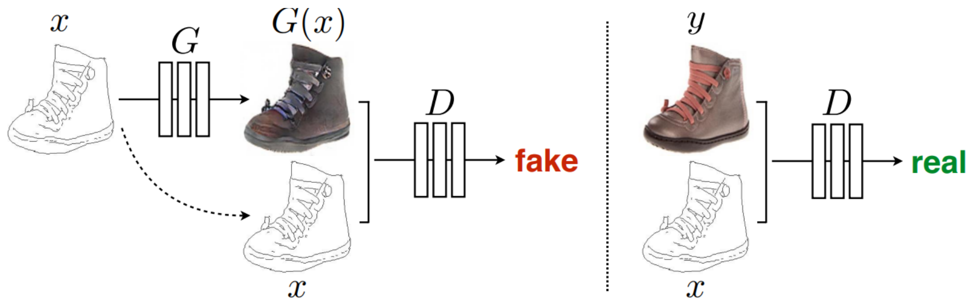



Figura 2.16: In questa immagine è possibile vedere un esempio di training di Pix2Pix per il task *edges to photo*, In tale configurazione il discriminatore apprende come discernere tra coppie di immagini e schizzi, e si può vedere come diversamente dal caso classico sia il generatore che il discriminatore ricevono in ingresso l'immagine di riferimento (lo schizzo) credits: Phillip Isola et al. [15]

.

### 2.4.2 L'architettura

L'architettura proposta per Pix2Pix utilizza dei blocchi convoluzionali ripresi dalla precedentemente discussa DCGAN [24], composti da tre componenti concatenate *Convolution-BatchNorm-Relu* per ogni layer, sia per il generatore che per il discriminatore. Per il generatore, nello specifico, è stata ripresa la struttura di U-Net, la quale è stata presentata ed utilizzata con successo per il task di *image segmentation* da parte di Olaf Ronneberger et al. [25] nel 2015. Tale struttura è in sostanza un *encoder* ed un *decoder* accoppiati con un *bottleneck* centrale, il quale funge da ponte tra le due metà e nel quale vengono estratte le informazioni globali dell'immagine, Il decoder in seguito utilizza queste informazioni per generare l'immagine di output. L'innovazione portata da U-Net è stata quella di aggiungere delle *skip connection* tra l'encoder e il decoder, in modo da poter utilizzare le informazioni globali estratte al livello del *bottleneck* e di unirle alle informazioni locali presenti nei layer dell'encoder, in modo tale da poter generare un'output più dettagliato e fedele all'immagine di riferimento. Di seguito possiamo vedere la Figura 2.17 che illustra sinteticamente le differenze tra un modello con struttura ad *encoder-decoder* e U-net.

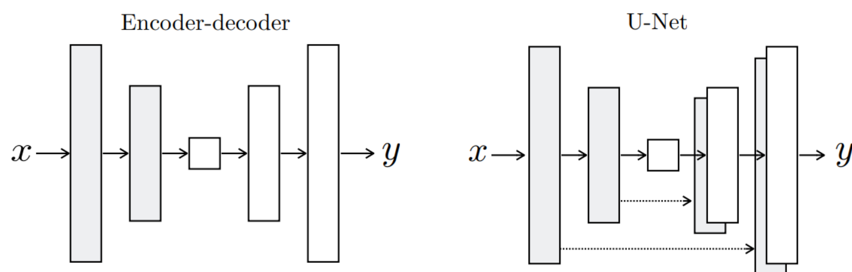



Figura 2.17: In questa immagine è illustrata intuitivamente la differenza tra un comune modello ad *encoder-decoder* e l'architettura di U-net.
credits: Phillip Isola et al. [15]

.

L'introduzione delle *skip-connections* nel modello U-Net è giustificata dal fatto che le

informazioni dell'immagine di input trovano difficoltà ad essere propagate efficacemente dopo una serie di *pooling layer*, dunque le *skip connection* permettono di bypassare questi layer e di poter utilizzare le informazioni locali presenti nel *encoder*. Nello stesso lavoro sono stati fatti dei test con e senza *skip connection*, e si è visto che l'aggiunta di queste ultime ha portato ad un miglioramento delle prestazioni del modello in diversi task. Un chiaro esempio è mostrato in figura 2.18, dove è possibile vedere una comparazione di un sample generato da 4 diversi modelli per il task *labels to scene*, questi quattro modelli alternano l'utilizzo o meno delle *skip connection* e l'utilizzo di una loss L1 a una loss L1 più *conditional adversarial loss*.

Figura 2.18: In questa immagine è possibile vedere una comparazione di un sample generato da 4 diversi modelli per il task *labels to scene*, i quattro esempi illustrano l'utilizzo o meno delle *skip connection* e l'utilizzo di una loss L1 a una loss L1 più *conditional adversarial loss*. credits: Phillip Isola et al. [15]

.

### 2.4.3 Alcuni esempi di applicazione

Di seguito sono mostrati alcuni esempi di applicazione del modello pix2pix, tra i quali: *colorization*, *edges to photo*, *aerial to map*, *labels to scene*, *labels to facade*, *day to night* e *edges to photo*.

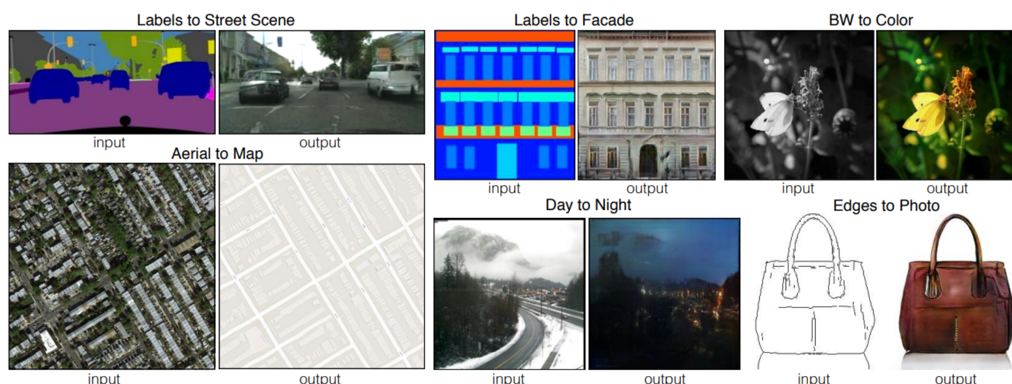



Figura 2.19: In questa immagine sono mostrati alcuni esempi di applicazione di pix2pix. credits: Phillip Isola et al. [15]

.

## 2.5 Large Mask inpainting with fast fourier convolution: LaMa

Arriviamo in fine al lavoro più recente presentato in questa tesi, e uno dei principali al quale il progetto che verrà illustrato successivamente è ispirato. LaMa (Large Mask inpainting with fast fourier convolution) è un modello proposto da Roman Suvorov et al. [28] nel 2021, tale architettura è stata proposta per il task dell'*inpainting*, il quale consiste nella ricostruzione di una porzione di immagine mancante, data la restante parte. LaMa si distingue dagli altri modelli proposti per l'inpainting per una caratteristica peculiare, l'utilizzo della *Fast Fourier Convolution* (FFC) [3], un operatore proposto da Lu Chi et al. nel 2020, la quale consente di elaborare informazioni globali dell'immagine sin dai primi layer mantenendo un costo computazionale simile a quello di una convoluzione standard con kernel di dimensioni ridotte. Un problema noto delle reti neurali convoluzionali è infatti il legame tra kernel size e receptive field, tanto più sarà grande il kernel size tanto più velocemente la rete sarà in grado di elaborare informazioni globali, tuttavia ciò comporta un aumento del costo computazionale che cresce in maniera esponenziale con l'aumentare delle dimensioni del kernel, al contrario la FFC si propone di risolvere questo problema ad un costo computazionale ridotto, infatti LaMa nei risultati presentati oltre a risultare mediamente migliore rispetto ai concorrenti, eccelle proprio per i suoi risultati su immagini in cui la parte da ricostruire è molto estesa.

### 2.5.1 Struttura del modello

In questa sezione vedremo la struttura del modello, analizzando l'input e la struttura dei blocchi principali che lo compongono, analizzando nello specifico il punto di forza di questa architettura, ovvero l'utilizzo della FFC.

L'input del modello è costituito da un'immagine $x$ e una maschera $m$, dove $m$ è una matrice binaria che ha dimensioni di $x$ ma un solo canale, e presenta valori 1 nelle posizioni in cui l'immagine è presente e 0 nelle posizioni in cui l'immagine deve essere ricostruita. Viene quindi effettuata un *element-wise multiplication* tra $x$ e $m$, e si concatena il risultato con $\overline{m}$, ottenendo il tensore di input $x' = [x \odot m, \overline{m}]$.

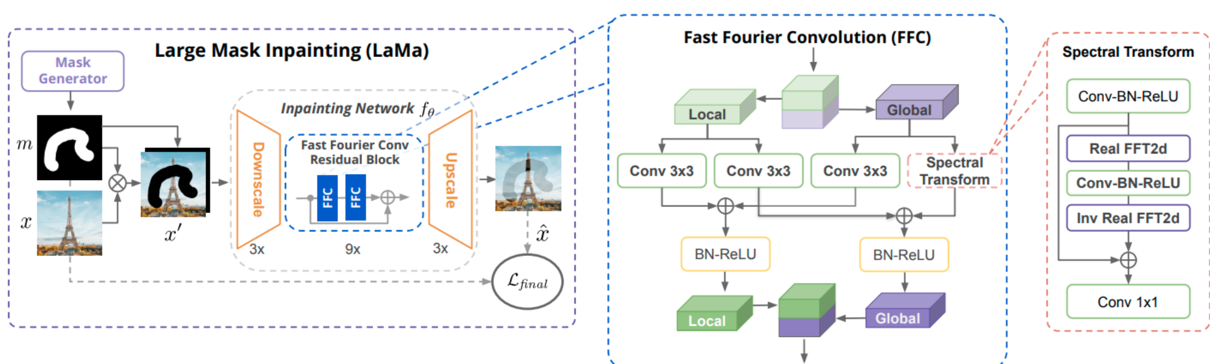



Figura 2.20: In questa immagine è illustrata schematicamente la struttura di LaMa. credits: Roman Suvorov et al. [28]

.

Un'ulteriore operazione che è possibile effettuare per aumentare la variabilità dell'output è quella di aggiungere rumore gaussiano alla matrice $\overline{m}$, considerando una matrice di rumore gaussiano $z$ con le stesse dimensioni di $\overline{m}$, procedendo come segue: $x' = [x \odot m, z \odot \overline{m}]$.

Il modello nella versione base con FCC è costituito inizialmente da 3 blocchi FFC con i seguenti valori stride 2, kernel size 3 e padding 1, ottenendo dunque per le equazioni [1.20, 1.21] un output di dimensioni dimezzate rispetto all'input su ogni layer, dunque ottenendo dopo i primi 3 blocchi un output di dimensioni $\frac{1}{8}$ rispetto all'input.
Dopo questa prima operazione di downsampling in cui comunque vengono utilizzati blocchi FCC, vengono aggiunti ulteriori 9 blocchi chiamati dall'autore *Fast Fourier Convolutional Residual Blocks*, costituiti semplicemente da 2 blocchi FCC con stride 1, kernel size 3 e padding 1, dunque mantenendo le dimensioni dell'input invariato, con in parallelo una connessione residuale che concatena l'input con l'output del secondo blocco FCC, è possibile osservare il collegamento tra i blocchi FCC nella Figura 2.20.
In fine seguono 3 layer di upsampling, attraverso *transposed convolution* con stride 2, kernel size 3 e padding 1, che riportano l'output alla dimensione originale, senza utilizzare blocchi FCC.

### Fast Fourier Convolution

A questo punto dopo aver snocciolato il contenuto del modello, scendiamo di un'ulteriore livello andando ad analizzare il contenuto dei blocchi FCC, i quali come visto sopra sono le componenti principali di questa architettura.
Prendendo come riferimento il rettangolo tratteggiato in blu in Figura 2.20, possiamo notare come il tensore in input al blocco FFC di dimensioni [N, C, H, W] venga diviso in due parti una viene utilizza per l'estrazione delle feature globali e una per l'estrazione delle feature locali, ovvero un blocco di dimensioni [N, C*(1-$\alpha$), H, W] e un blocco di dimensioni [N, C*$\alpha$, H, W], dove $\alpha$ è un valore compreso tra 0 e 1, e rappresenta la percentuale di canali che vengono utilizzati per ognuna delle due componenti.

| **Label** | **Operazione** | **Dimensioni ingresso uscita** |
|---|---|---|
| **a** | **Convolution1x1 ∘ BatchNorm ∘ ReLU** | $\mathbb{R}^{C\times H\times W} \rightarrow \mathbb{R}^{C\times H\times W}$ |
| **b** | **Real FFT2d** | $\mathbb{R}^{C\times H\times W} \rightarrow \mathbb{C}^{C\times H\times \frac{W}{2}+1}$ |
| **c** | **Complex to real** | $\mathbb{C}^{C\times H\times \frac{W}{2}+1} \rightarrow \mathbb{R}^{2C\times H\times \frac{W}{2}+1}$ |
| **d** | **Convolution1x1 ∘ BatchNorm ∘ ReLU** | $\mathbb{R}^{2C\times H\times \frac{W}{2}+1} \rightarrow \mathbb{R}^{2C\times H\times \frac{W}{2}+1}$ |
| **e** | **Real to complex** | $\mathbb{R}^{2C\times H\times \frac{W}{2}+1} \rightarrow \mathbb{C}^{C\times H\times \frac{W}{2}+1}$ |
| **f** | **Inverse Real FFT2d** | $\mathbb{C}^{C\times H\times \frac{W}{2}+1} \rightarrow \mathbb{R}^{C\times H\times W}$ |
| **g** | **Concat(a, f)** | $\mathbb{R}^{C\times H\times W} \rightarrow \mathbb{R}^{2C\times H\times W}$ |
| **h** | **Convolution1x1 ∘ BatchNorm ∘ ReLU** | $\mathbb{R}^{2C\times H\times W} \rightarrow \mathbb{R}^{2C\times H\times W}$ |

Tabella 2.1: Operazioni che compongono il sotto blocco *Spectral Transformation*.

A questo punto il tensore contenente le informazioni locali (in verde) viene sottoposto a due blocchi convoluzionali standard, mentre il tensore contenente le informazioni globali (in viola) viene sottoposto ad una operazione di convoluzione standard e ad una operazione detta di *Spectral Transformation*, quest'ultima nello specifico presenta un'operazione in sequenza un operazione di convoluzione standard seguitada un layer per la batch normalization e un layer di attivazione ReLU, successivamente viene applicata la *Real Fast Fourier Transform* (RFFT), la quale è una trasformata di Fourier che opera solo su metà dello spettro. Dopo la trasformazione nel dominio della frequenza, viene applicato un ulteriore layer di convoluzione (*Convolution-BatchNorm-ReLU*) e successivamente viene applicata la *Inverse Real Fast Fourier Transform* (IRFFT), che è l'inversa della RFFT, e permette di riportare il tensore a valori reali. Le operazioni del sotto blocco di *Spectral Transformation* sono riassunte nella tabella 2.1.

### 2.5.2 La loss function

L'addestramento di un modello di inpainting porta con se una serie di sfide complesse, prima fra tutte l'ambiguità del task, infatti data l'area di un'immagine da rigenerare i possibili riempimenti sono molto numerosi, e non è possibile stabilire a priori quale sia la soluzione migliore, specialmente nel caso di aree da rigenerare di elevata dimensione rispetto all'immagine, ciò rende difficile la definizione di una loss efficacie.

#### La Perceptual Loss (PL) e la High receptive field perceptual loss (HRFPL)

Precedenti tentativi di altri autori di utilizzare loss supervisionate per l'inpainting o la generazione di immagini hanno portato a risultati non soddisfacenti, in quanto queste loss tipicamente forzano il generatore a ricostruire l'immagine originale in maniera esatta, questo metodo infatti porta il generatore a generare immagini sfocate che sono la media delle possibili soluzioni presentate nel dataset di addestramento.

Un'alternativa alle tradizionali loss supervisionate è data dalla *perceptual loss* proposta per i task di *style transfer* e *super-resolution* da Justin Johnson et al. [16]. Questa particolare loss sfrutta un modello pre-addestrato $\phi$ per l'estrazione delle feature, in questo modo è possibile definire una loss che misuri la distanza tra le feature di tale modello, dunque in uno spazio semanticamente più significativo rispetto allo spazio dei pixel, tale proprietà consente alla loss di valutare la somiglianza semantica tra due immagini, piuttosto che la somiglianza pixel per pixel, permettendo al modello di ricevere un feedback molto più veritiero sulla bontà del suo output, nonostante questo non sia identico all'immagine originale.

Come ben noto all'interno dei modelli convoluzionali si va a creare una stratificazione di feature in cui quelle di basso livello (vicine all'input) sono associate a caratteristiche locali dell'immagine, mentre quelle di alto livello (vicine all'output) sono associate a caratteristiche globali come forme o comunque strutture più complesse. Per tale ragione tale loss può essere anche influenzata per dare più peso a caratteristiche globali o locali.

In generale la perceptual loss è definita in [16] come:

$$\mathcal{L}_{PL_\phi}(\hat{y}, y) = \sum_{i=1}^{N} \frac{1}{C_i H_i W_i} \sum_{c=1}^{C_i} \sum_{h=1}^{H_i} \sum_{w=1}^{W_i} \left(\phi_i(\hat{y})_{c,h,w} - \phi_i(y)_{c,h,w}\right)^2 \tag{2.15}$$

Tale implementazione della perceptual loss calcola la distanza $L_2$ layer per layer del modello $\phi$ tra l'immagine ricostruita $\hat{y}$ e l'immagine originale $y$ al quadrato, in tal modo si ottengono $n$ distanze $L_2$ una per ogni layer del modello $\phi$, successivamente si effettua la somma di tali distanze e si ottiene la perceptual loss. Una possibile alternativa consiste nell'effettuare la media delle distanze $L_2$ invece della somma, comme mostrato nella Figura 2.21 presa da [31].

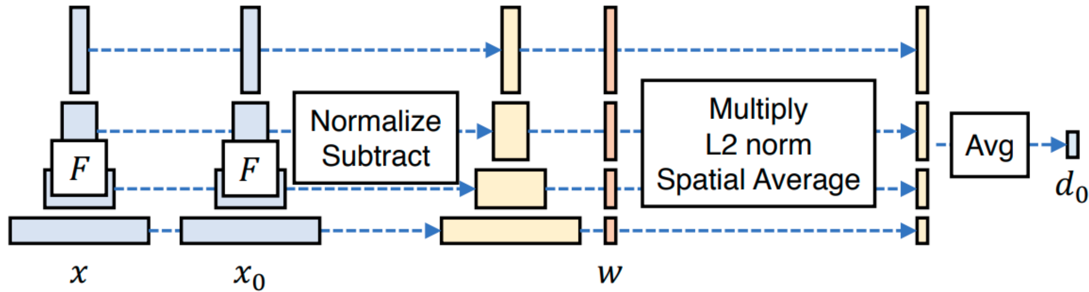



Figura 2.21: In questa immagine sono mostrati schematicamente i passaggi che permettono di calcolare la perceptual loss da un modello convoluzionale.
credits: Richard Zhang et al. [31]

Per LaMa è stata proposta una variante della PL, denominata *High receptive field perceptual loss* (HRFPL), la quale presenta le medesime caratteristiche appena descritte per la classica PL ma con la differenza che il modello $\phi$ utilizzato per il calcolo delle feature presenta delle caratteristiche particolari, ovvero un campo ricettivo esteso come dice il nome stesso (HRF), tale proprieta può essere ottenuta utilizzando un modello che sfrutti la *dilated convolution* o la *Fast Fourier Convolution* (FFC) precedentemente illustrata.

#### L'adversarial loss

Un'ulteriore loss utilizzata per l'addestramento di LaMa è la *non-saturating adversarial loss*, utilizzata nella prima versione del modello GAN del 2014, con adeguati accorgimenti per il caso dell'inpainting.
In particolare è stata ripresa la struttura del *PatchGAN discriminator* proposto per Pix2Pix in [15], Tale approccio utilizza un discriminatore che viene applicato a patch di dimensioni prefissate, utilizzando la tecnica dello *sliding window*. Diversamente da quando avveniva per Pix2Pix, in LaMa il discriminatore nelle immagini su cui viene effettuato l'inpainting non considera ogni patch come un esempio "falso", ma considera false solo le patch che contengono un'area rigenerata.

Consideriamo a questo punto il generatore $G$ definito dai parametri $\theta$ e il discriminatore $D$ definito dai parametri $\xi$, un'immagine $x$ e la sua maschera binaria $m$, l'immagine $x' = stack(x \odot m, \overline{m})$ è l'input del generatore $G$ (dove $\odot$ indica il prodotto elemento per elemento), l'immagine $\hat{x} = G_\theta(x, m)$ è l'immagine ricostruita definita come $\hat{x} = G_\theta(x')$.

Abbiamo dunque l'adversarial loss $\mathcal{L}_{adv}$ definita come la somma delle loss del generatore $\mathcal{L}_G$ e del discriminatore $\mathcal{L}_D$:

$$\mathcal{L}_D = -\mathbb{E}_x [\log D_\xi(x)] - \mathbb{E}_{x,m} [\log (D_\xi(\hat{x})) \odot \overline{m}] - \mathbb{E}_{x,m} [\log (1 - D_\xi(\hat{x})) \odot m] \tag{2.16}$$

$$\mathcal{L}_G = -\mathbb{E}_{x,m} [\log D_\xi(\hat{x})] \tag{2.17}$$

$$\mathcal{L}_{adv} = \mathcal{L}_D + \mathcal{L}_G \tag{2.18}$$

Si noti che per $\mathcal{L}_D$ i gradienti vengono propagati soltanto attraverso il discriminatore, mentre per $\mathcal{L}_G$ i gradienti vengono propagati fino al generatore, ma non vengono applicati al discriminatore. Un'altro fatto importante che va considerato nell'equazione 2.16 è che il termine $\log (D_\xi(\hat{x})) \odot \overline{m}$ viene applicato soltanto alle patch dell'immagine ricostruita $\hat{x}$ che non presentano parti rigenerate, mentre il termine $\log (1 - D_\xi(\hat{x})) \odot m$ viene applicato soltanto alle patch dell'immagine ricostruita che presentano parti rigenerate.

#### La regolarizzazione R1

Un'importante accorgimento che è stato introdotto per migliorare la stabilità dell'addestramento di LaMa è l'R1 regularization, proposta da Lars Mescheder et al. in [21], nel 2018.
In questo articolo Mescheder et al. hanno investigato il problema dell'instabilità dell'addestramento dei modelli GAN, studiano un caso estremamente semplificato la *Dirac-GAN* in cui il generatore $G$ è un modello definito da un solo parametro, appunto rappresentante una delta di Dirac, e il discriminatore $D$ è un modello che deve distinguere tra la delta di Dirac di riferimento e la delta di Dirac generata. L'esperimento per quanto banale ha permesso di evidenziare l'incapacità del modello di convergere alla sovrapposizione delle due Delta, ovvero la condizione di Equilibrio di Nash, in quanto nel punto di convergenza il gradiente del discriminatore non è nullo e dunque spinge via il generatore da tale punto di equilibrio, il quale rappresenterebbe la soluzione ottima per il problema.

In tale lavoro sono state confrontate diverse tecniche di regolarizzazione per risolvere il problema, come la *WGAN-GP* o *l'instance noise regularization* e altre, ma in particolare è stata proposta la *R1 regularization*, questa regolarizzazione ha dimostrato di riuscire a risolvere il problema, in un modo migliore rispetto ad altri metodi, ma sopratutto utilizzando una quantità contenuta di risorse computazionali. Di seguito la formulazione matematica della *R1 regularization*:

$$\mathcal{L}_{R1} = \frac{\gamma}{2} \mathbb{E}_{x \sim p_{data}(x)} \left[ \|\nabla_x D_\xi(x)\|_2^2 \right] \tag{2.19}$$

Dietro questa equazione si nasconde un concetto tanto semplice quanto efficacie, infatti l'obbiettivo della *R1 regularization* è proprio quello di ridurre il gradiente del discriminatore nei pressi del punto di equilibrio, in modo da evitare che il generatore venga spinto via da tale

punto.

Per ottenere questo risultato con la *R1* si applica una penalità al discriminatore come componente della loss totale, tale penalità come si può vedere dall'equazione 2.19 è proporzionale alla media del quadrato della norma del gradiente del discriminatore, per i sample reali. Infatti si parla di $x \sim p_{data}(x)$, ovvero si considerano tutti i sample reali, e non quelli generati dal generatore, in quanto l'obbiettivo è minimizzare il gradiente del discriminatore nei pressi del punto di equilibrio, che corrisponde alla distribuzione reale dei dati.

Questa strategia permette di ottenere un risultato ottimo comparato ad altre tecniche come è possibile vedere nella Figura 2.22, vediamo infatti un'ulteriore esempio in cui una semplice gan deve generare sample di una distribuzione bidimensionale, la quale è a forma di anello. Nella figura i punti blu rappresentano i sample reali, mentre i punti arancioni i sample generati dalla gan, mentre sullo sfondo da viola (min) a giallo(max) è rappresentata l'intensità del gradiente del discriminatore per i vari punti dello spazio bidimensionale, si può notare come il gradiente per il caso in cui è stata utilizzata la R1 o la R2 (una variante della R1) sia quasi nullo rispetto agli altri casi.

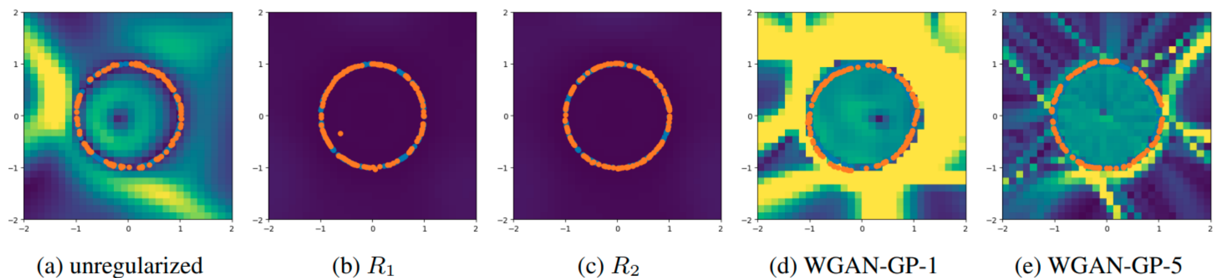



Figura 2.22: In questa immagine sono mostrati i risultati ottenuti da Mescheder et al. in [21] utilizzando la R1 regularization. In blu è rappresentata la distribuzione di riferimento, in arancione i campioni generati, mentre da viola a giallo si ha l'intensità del gradiente di $D$.

#### La loss finale

Vediamo in fine la loss finale che si compone come la somma delle componenti descritte in precedenza, ovvero la *High receptive field loss* $\mathcal{L}_{HRF}$, la *Adversarial loss* $\mathcal{L}_{adv}$ e la *R1 regularization* $\mathcal{L}_{R1}$. Oltre a queste componenti vi è una ulteriore loss che è stata utilizzata, la quale ha il medesimo funzionamento della *Perceptual loss*, ma utilizza il le features del discriminatore invece che quelle del modello addestrato su ImageNet, tale loss è stata chiamata *Discriminator-based perceptual loss* $\mathcal{L}_{DiscPL}$.
Di seguito la formulazione matematica della loss finale:

$$\mathcal{L}_{final} = \kappa\mathcal{L}_{adv} + \alpha\mathcal{L}_{HRFPL} + \beta\mathcal{L}_{DiscPL} + \gamma\mathcal{L}_{R1} \tag{2.20}$$

Si noti che ogni loss è provvista di un coefficiente moltiplicativo che permette di bilanciare le varie componenti, in modo da ottenere un risultato ottimale, in base al dataset o in base ad altri fattori.

### 2.5.3 Alcuni risultati

Sono mostrati in seguito alcuni risultati ottenuti da LaMa, in cui è possibile osservare come il modello sia in grado di ricostruire porzioni di immagine molto estese, e di come l'utilizzo della FFC dia al modello delle eccezionali capacità di ricostruire strutture periodiche, come ad esempio le finestre di un edificio, o i fori sulla lamiera di una serranda.

Figura 2.23: In questa immagine sono mostrati alcuni risultati ottenuti da LaMa. In blu sono rappresentate le maschere applicate alle immagini, e dunque corrispondenti alle arre che sono state rimosse prima della propagazione nella rete per la ricostruzione. Le immagini sulla destra rappresentano l'output del modello.
credits: Roman Suvorov et al. [28]
.

# Capitolo 3

# Materiali e metodi

## 3.1 Il Dataset: Severstal steel defect detection

L'acciaio è uno dei materiali più comunemente utilizzati in tutto il mondo, e la sua produzione e il suo costo sono in continua crescita, come è possibile vedere dal report rilasciato nel 2022 dalla World Steel Association [2], un'organizzazione *no-profit* impegnata nello sviluppo di questo settore. Tale documento mostra come la produzione di acciaio sia stata in continua crescita negli ultimi 70 anni, e come la produzione mondiale di acciaio abbia raggiunto le 1.95 miliardi di tonnellate nel 2021 in crescita del 3.8% rispetto al 2020, e si prevede che questa crescita continui nei prossimi anni, salvo momentanee battute d'arresto dovute a crisi economiche o pandemie. La sua versatilità e la sua resistenza lo rendono un materiale molto utilizzato in diversi settori, come l'edilizia, l'automotive, l'industria elettronica, l'industria aerospaziale, ecc. Per produzioni su larga scala di acciaio come di altri materiali o prodotti, è necessario che il materiale sia di qualità, e che non contenga difetti, ma è difficile per gli operatori umani rilevare difetti come graffi, crepe, ecc. con elevata affidabilità, per tale ragione è necessario adottare sistemi automatizzati che siano in grado di rilevare difetti in modo affidabile e veloce.

L'automazione del controllo qualità è stata la scelta fatta da Severstal, una delle principali aziende produttrici di acciaio in Russia, che produce circa 10 milioni di tonnellate di acciaio all'anno. Severstal ha messo a disposizione **Severstal steel defect detection dataset** nel 2019 per permettere a chiunque di testarvi le proprie idee, sperando di riuscire ad aumentare l'affidabilità del proprio sistema di rilevazione difetti sfruttando la comunità di Kaggle. Il dataset è disponibile al seguente link: https://www.kaggle.com/c/severstal-steel-defect-detection/data.

Per la competizione è stato rilasciato esclusivamente il training set, dunque in questo progetto tale dataset è stato ulteriormente diviso in training e test set. Il training set originale è composto da 12568 immagini, che verranno divise in un 50% per il nuovo training set e 50% per il nuovo test set, quindi un totale di 6284 immagini per il training set e 6284 immagini per il test set. Questo numero è stato scelto in modo da garantire un numero minimo di immagini per effettuare l'addestramento del modello, in quanto lo scopo non è addestrare un buon modello ma valutare l'efficacia del metodo per dataset di piccole dimensioni.

Il training set è stato poi utilizzato per il training del generatore di difetti sintetici, mentre il

test set per valutare la qualità dei dati generati tramite FID score.

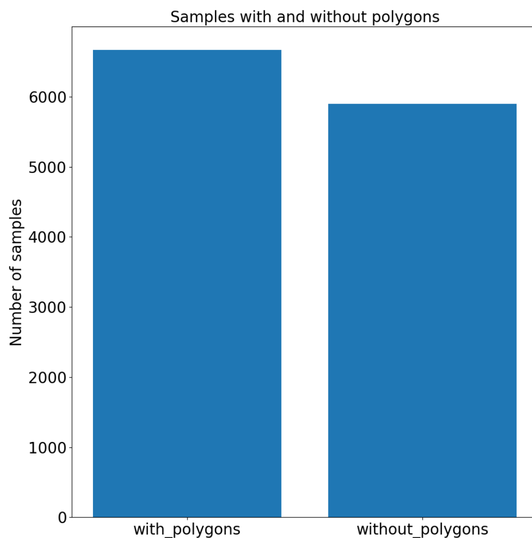
### 3.1.1 Distribuzione del dataset

In questa sezione viene presentata un'analisi tecnica della distribuzione del dataset, considerando il numero di immagini, numero e distribuzione dei poligoni in esse, per comprendere nel dettaglio il dataset e la sua composizione.

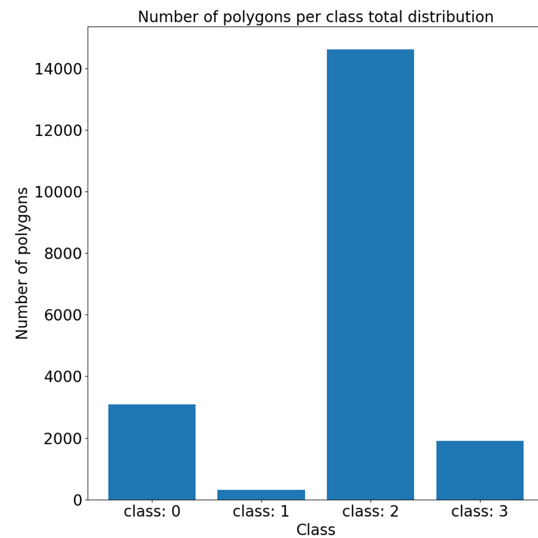
Il Severstal dataset è composto come già detto da 12568 immagini di lastre di acciaio con risoluzione 256x1600, di queste immagini abbiamo 6666 immagini con difetti, e 5902 immagini senza difetti. Il severstal dataset presenta 4 classi di difetti, le quali purtroppo, come spesso accade nei dataset provenienti da casi reali, non sono bilanciate, infatti abbiamo:

| | **Classe 1** | **Classe 2** | **Classe 3** | **Classe 4** |
|---|---|---|---|---|
| **Numero di poligoni** | 3082 | 321 | 14622 | 1902 |

Tabella 3.1: Distribuzione delle immagini per classe.

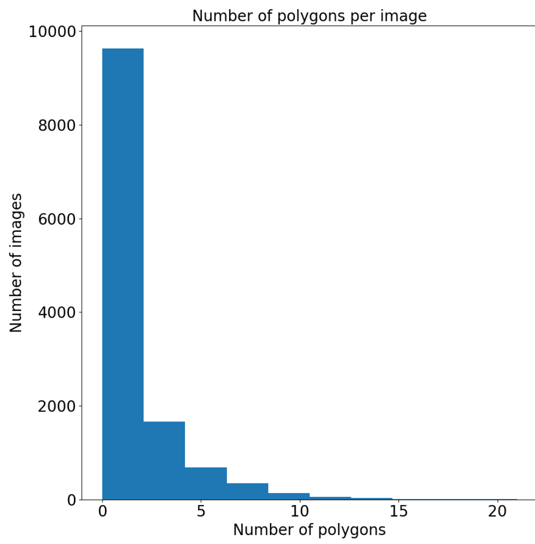


(a) Distribuzione delle immagini con e senza difetti.

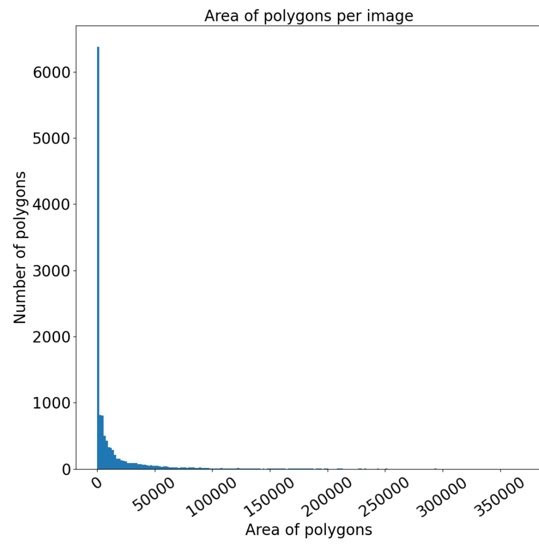


(b) Distribuzione dei poligoni per classe.

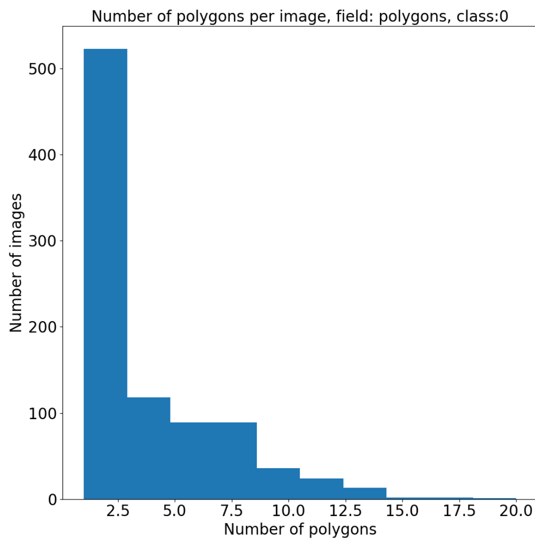
Altri parametri interessanti che vanno presi in considerazione riguardano la distribuzione dei poligoni nelle immagini, ovvero la distribuzione del numero di poligoni in ogni immagine, per avere un'idea della frequenza con cui questi sono presenti. Una valutazione è stata fatta anche per la distribuzione dell'area dei poligoni, in quanto ai fini della generazione dei difetti sintetici è importante conoscere la quantità di pixel relativa ai difetti a disposizione, consideriamo infatti che 10 difetti con area di 100 pixel (1000 pixel) portano con se molta meno informazione di 1 difetto con area di 4000 pixel. Le stesse valutazioni sono poi state fatte per ogni classe di difetti, in quanto ogni classe ha una sua specifica distribuzione.



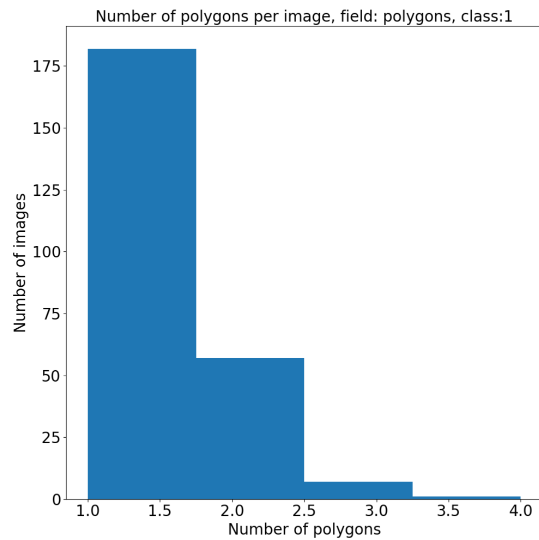
(a) Distribuzione numero poligoni per immagine.



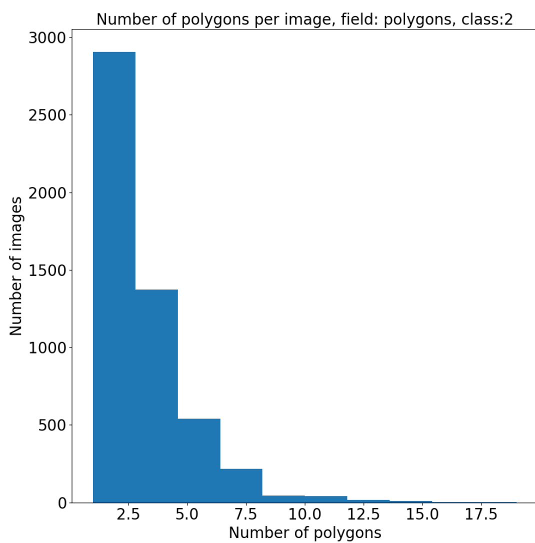
(b) Distribuzione area poligoni per immagine.



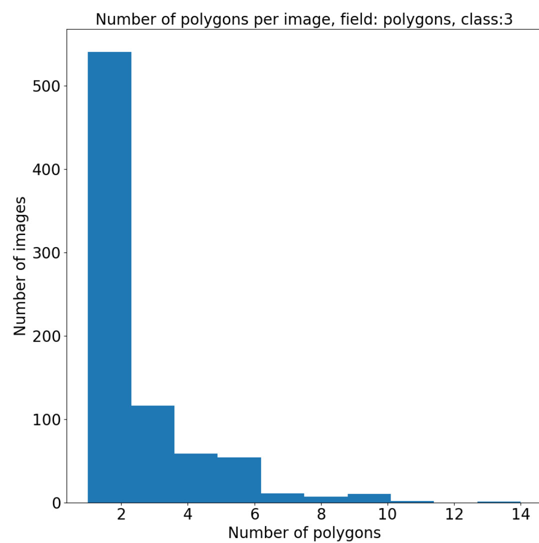
(c) Distribuzione numero poligoni classe 1 per immagine



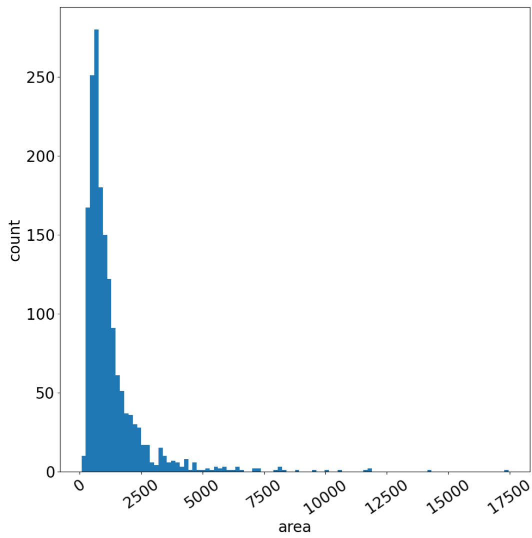
(d) Distribuzione numero poligoni classe 2 per immagine



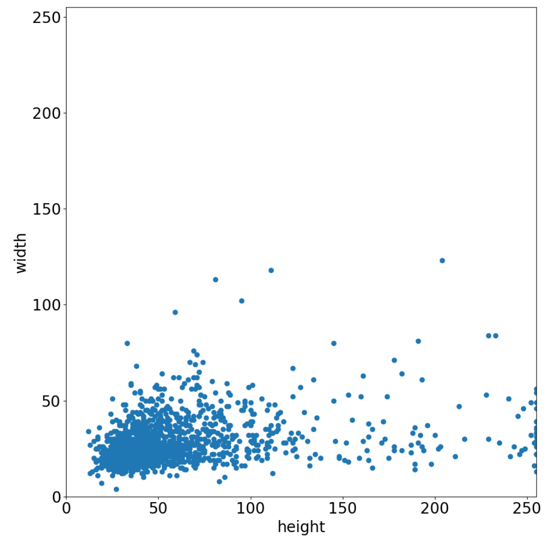
(e) Distribuzione numero poligoni classe 3 per immagine



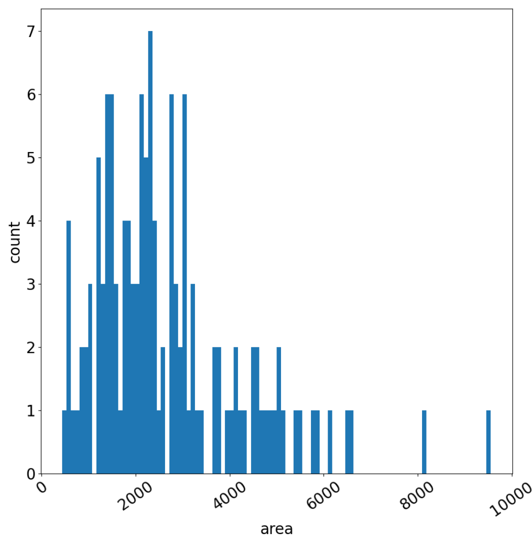
(f) Distribuzione numero poligoni classe 4 per immagine



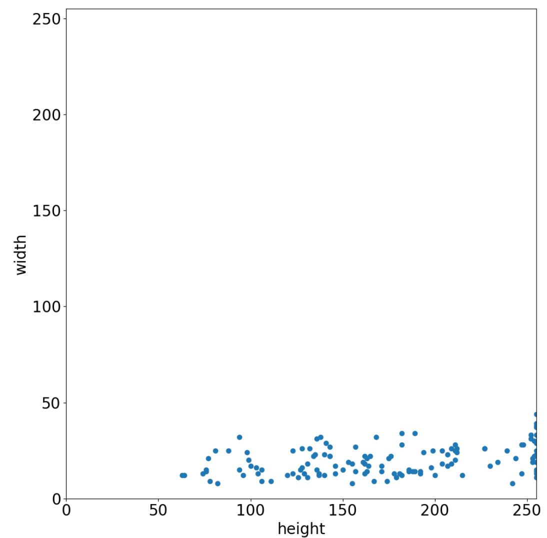

(a) Distribuzione area poligoni classe 1.



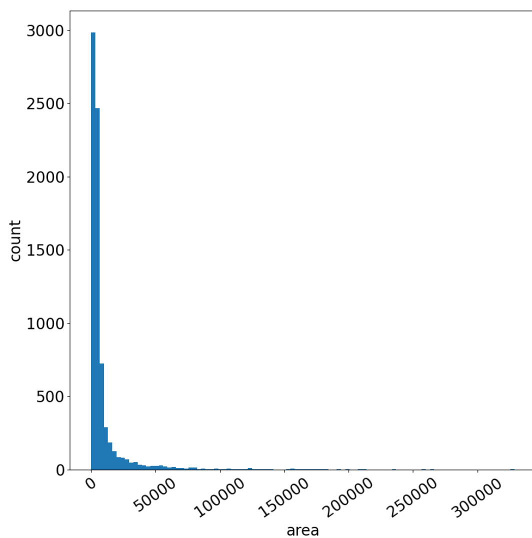

(b) Distribuzione aspect-ratio poligoni classe 1.



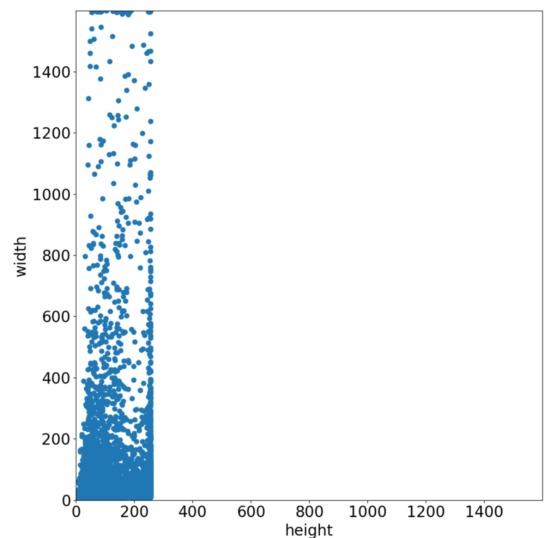

(a) Distribuzione area poligoni classe 2.



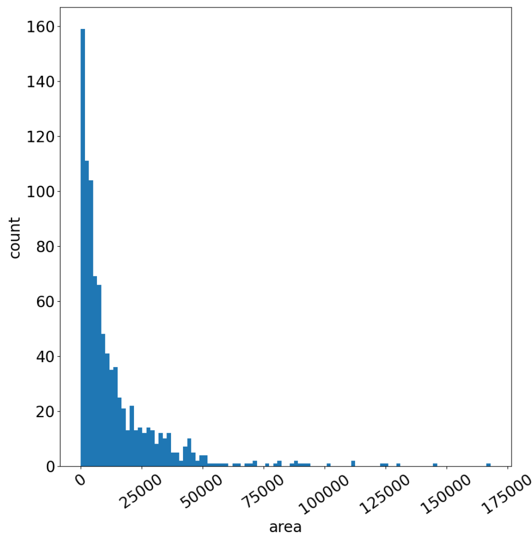

(b) Distribuzione aspect-ratio poligoni classe 2.



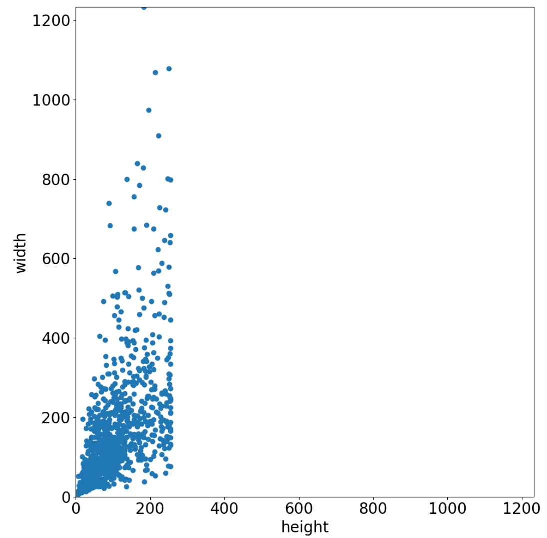

(a) Distribuzione area poligoni classe 3.




(b) Distribuzione aspect-ratio poligoni classe 3.

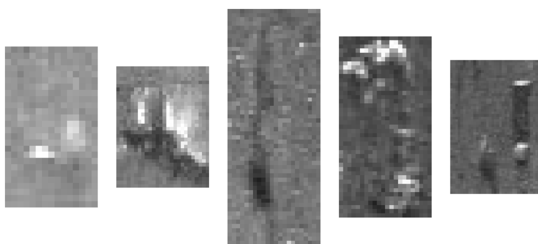



(a) Distribuzione area poligoni classe 4.

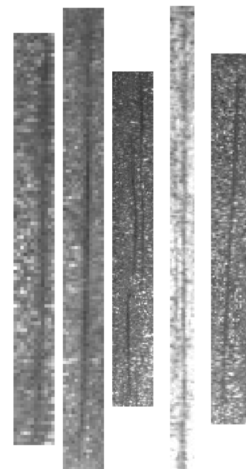



(b) Distribuzione aspect-ratio poligoni classe 4.

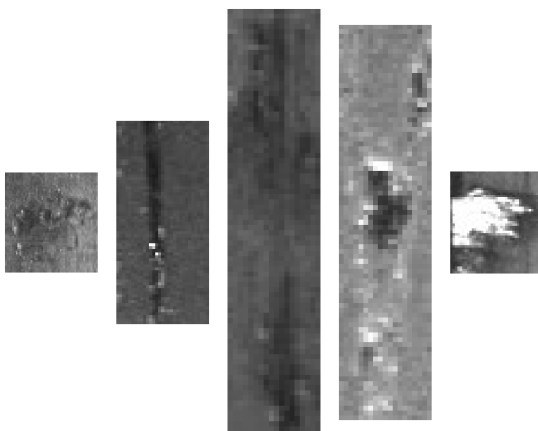
Vediamo di seguito alcuni esempi di difetti estratti dal dataset tramite crop, appartenenti alle quattro classi. Con il dataset non vengono fornite informazioni riguardo alle caratteristiche delle 4 classi ma è possibile dedurre alcune informazioni analizzando le immagini e le distribuzioni mostrate. Si noti che gli esempi mostrati di seguito in alcuni casi sono stati ridimensionati o ruotati per una migliore illustrazione.



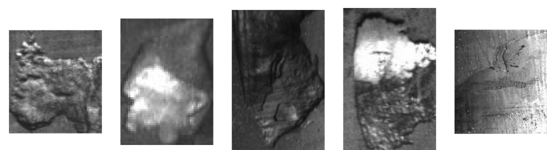
(a) Esempi di difetti classe 1.



(b) Esempi di difetti classe 2.



(c) Esempi di difetti classe 3.



(d) Esempi di difetti classe 4.

## 3.2 Librerie e Framework

In questo progetto è stato utilizzato il linguaggio Python che ad oggi è uno dei linguaggi più utilizzati per il machine learning e l'intelligenza artificiale, data la sua semplicità e la sua versatilità, per non parlare della vasta scelta di librerie che offre. Nonostante Python sia un linguaggio interpretato, è possibile utilizzarlo per applicazioni che richiedono un alto livello di performance, in quanto le librerie che si occupano di effettuare le operazioni più pesanti, sono scritte in C/C++, e vengono utilizzate tramite Python bindings, che permettono di utilizzare le librerie scritte in C/C++ come se fossero state scritte in python, è questo il caso di librerie come Numpy, OpenCV, PyTorch, ecc.

### 3.2.1 Numpy

Una delle librerie più importanti per Python è Numpy, che permette di lavorare con array multidimensionali, e di effettuare operazioni su di essi in modo efficiente con un'interfaccia semplice e intuitiva. Questa libreria benché non sia integrata in Python, è una delle librerie più utilizzate ed è uno standard de facto per il calcolo scientifico per svariate librerie per Python. La documentazione e informazioni aggiuntive sulla libreria sono disponibili sul sito https://numpy.org/.

### 3.2.2 OpenCV

OpenCV è una libreria open source per il computer vision, scritta in C++, ma utilizzabile anche in Python tramite Python bindings. Ad oggi è una delle librerie più utilizzate per l'elaborazione di immagini e video, in quanto offre un'ampia gamma di funzionalità. In questo progetto è stata utilizzata per effettuare diverse operazioni di pre-elaborazione del dataset, e per lo script demo di inferenza. La documentazione e informazioni aggiuntive sulla libreria sono disponibili sul sito https://opencv.org/.

### 3.2.3 PyTorch

PyTorch è un framework open source per il deep learning, sviluppato da Facebook, che permette di effettuare operazioni su tensori in modo efficiente sfruttando le GPU, e di effettuare automaticamente il backpropagation, rendendo l'operazione di definire una rete neurale e addestrarla molto semplice e veloce. La documentazione e informazioni aggiuntive sulla libreria sono disponibili sul sito https://pytorch.org/.

### 3.2.4 Distributed data parallel

Distributed data parallel non è una libreria a se stante ma un modulo integrato nel framework PyTorch, che permette di scalare il training di un modello neurale su più GPU in un singolo nodo, o su più nodi. DDP parallelizza il training generando un processo separato per ogni

GPU, ognuno presenta una copia identica del modello, durante il training la retro-propagazione dell'errore innesca un *hook* che effettua la sincronizzazione dei gradienti mediante delle primitive di sincronizzazione tra i processi, in tal modo si ottiene un unico gradiente aggiornato per tutte le istanze.

## 3.3 Google cloud compute instance

L'addestramento del modello neurale è stato effettuato mediante l'utilizzo dei servizi di Google Cloud Platform, in particolare è stata utilizzata una macchina virtuale **n1-standard-8** con le seguenti caratteristiche:

- **Sistema Operativo:** Ubuntu 18.04
- **CPU:** 8 vCPU
- **GPU:** 2x Nvidia Tesla T4 (16 GB)
- **RAM:** 32 GB

Questa tipologia di macchina virtuale è una delle consigliate per l'addestramento di modelli di deep learning, in quanto permette di utilizzare le GPU T4 di Nvidia, le quali ad oggi costituiscono un ottimo compromesso tra costo e performance. Infatti l'affitto di una macchina virtuale con queste caratteristiche, con ben 2 Tesla T4, per un mese ha un costo di circa 500 euro, mentre una macchina virtuale con una sola GPU come una V100 o una A100 può superare tranquillamente i 3000€.

Questa configurazione non garantisce le stesse performance delle controparti di fascia alta ma comunque consentono di avere a disposizione una potenza di calcolo discreta e un quantitativo di memoria di ben 32 GB di memoria GDDR6, che è più che sufficiente per l'addestramento di modelli neurali di modeste dimensioni se pur con dimensioni di batch ridotte.

## 3.4 Repository del progetto

Il codice sorgente del progetto è disponibile su GitHub al seguente indirizzo:

https://github.com/MassimilianoBiancucci/COIGAN-controllable-object-inpainting

Nel repository è presente il codice effettivo per eseguire un training, il codice per effettuare la preparazione del dataset, il quale necessita di un formato particolare, e lo script per effettuare una valutazione interattiva del modello. Tutti i passaggi necessari per preparare l'ambiente, e sistemare i file di configurazione sono descritti nel file README.md.

# Capitolo 4

# Sviluppo del progetto

In questa sezione verrà illustrato lo sviluppo del progetto, seguendo le varie fasi che hanno portato alla realizzazione del modello per la generazione condizionata di immagini con difetti, partendo da una descrizione generale della pipeline di addestramento e delle sue componenti, per poi passare alla descrizione delle varie fasi di sviluppo, con particolare attenzione alla fase di preparazione dei dati e alla fase di addestramento del modello. Verrà inoltre discussa la valutazione del modello mediante FID.

## 4.1 Definizione della pipeline di addestramento

La pipeline di addestramento per la generazione dei difetti realizzata in questo progetto è stata principalmente ispirata dalla pipeline di LaMa, alla quale però è stato necessario apportare diverse modifiche per poter ottenere il risultato desiderato. Il task effettuato da LaMa infatti si basa sulla ricostruzione di parte dell'immagine originale, alla quale viene cancellata una certa porzione prima di passarla al generatore, mentre il task che si vuole effettuare in questo progetto è quello di modificare l'immagine aggiungendo delle caratteristiche appartenenti ad una o più distribuzioni. Come visto precedentemente, il modello di LaMa è stato addestrato prendendo in input un'immagine con una certa porzione rimossa, concatenata con una maschera che indica la posizione della porzione rimossa, alla quale eventualmente viene aggiunto del rumore gaussiano per favorire la generalizzazione del modello e una maggiore variabilità dell'output. Per quanto riguarda COIGAN il condizionamento dell'output avviene nella medesima maniera, ovvero concatenando delle maschere all'immagine, diversamente da LaMa le aree corrispondenti alle maschere e dunque alla porzione di immagine sulla quale vanno applicati i difetti non viene rimossa, in quanto quest'ultima contiene delle informazioni utili per la generazione di un output più coerente con l'immagine originale.
Di seguito è rappresentato un'esempio di input, con i primi tre canali del tensore di ingresso rappresentanti l'immagine base, mentre i successivi 4 canali rappresentano le maschere per il condizionamento dell'output.

Figura 4.1: Figura che illustra un esempio di input di COIGAN, il quale è un tensore di 7 canali, in questo caso con un'immagine RGB e 4 canali per le maschere che effettuano il condizionamento della generazione dei difetti.

Nella Pipeline di LaMa come visto nella sezione 2.5, ci sono 3 loss principali, esclusa la regolarizzazione, le quali sono: la loss L1, la *Adversarial loss* e la *Perceptual loss*. La loss L1 è una componente della loss che spinge il generatore verso il centro della distribuzione dei dati, la quale anche se come illustrato da Goodfellow può portare a risultati non ottimali, per quanto visto nella pubblicazione di Pix2Pix, se utilizzata con il giusto peso in congiunzione con l'*Adversarial loss* sembra migliorare i risultati generali. La *Adversarial loss* è la componente principale della loss, che spinge il generatore a generare immagini che abbiano delle caratteristiche appartenenti alla distribuzione del *training set*. In fine la *Perceptual loss* è una componente che spinge il generatore a completare l'immagine con delle caratteristiche che siano coerenti con quelle della parte rimossa dell'immagine originale, senza bisogno di effettuare un match perfetto, quest'ultima applicata con le *features* di un modello preaddestrato e attraverso quelle estratte dal discriminatore.

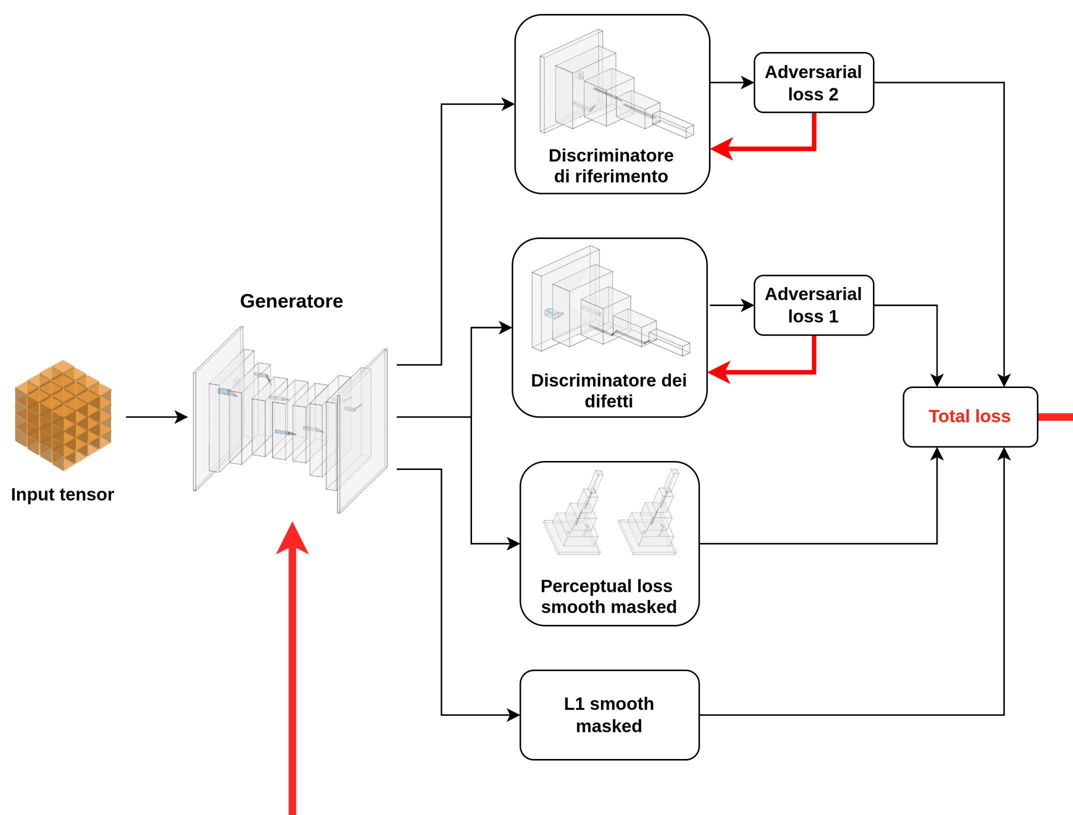



Figura 4.2: Figura che illustra in maniera riassuntiva le componenti principali della pipeline di addestramento di COIGAN. Le frecce rappresentano la propagazione dei vari tensori, mentre le linee rosse indicano la propagazione dei gradienti delle loss.

Per quanto riguarda COIGAN, le modifiche principali che sono state apportate sono relative a come sono state utilizzate le adversarial loss, e all'introduzione di una *smooth mask* per pesare la loss L1 e la *Perceptual loss*. In oltre ci sono delle modifiche sostanziali nella preparazione dei dati, i quali in questo caso non vengono utilizzati direttamente come input del generatore, applicando semplicemente una maschera per rimuovere una certa porzione. Nella Figura 4.2 è riassunta la pipeline di addestramento, per la quale in seguito verranno approfondititi i vari componenti.

## 4.2 Preparazione dei dati per la pipeline

In questa sezione verrà illustrato come è stato preparato il dataset per la pipeline di addestramento, partendo dal dataset Severstal steel defect detection dataset, nel suo formato originale, per poi passare alla conversione del dataset nel formato *line json*, utilizzato poi anche dal dataloader della training pipeline. Verrà illustrato come è stato effettuato un *fair split* del dataset in train e test set, che tiene conto della distribuzione dei difetti e di come sono stati creati i dataset degli oggetti, delle immagini base e dei dataset utilizzati come riferimento per l'addestramento del modello. Tutte queste operazioni vengono effettuate dallo script `scripts/prepare_severstal_dataset.py`.

### 4.2.1 Severstal steel defect detection dataset reader

Il dataset *Severstal steel defect detection* è composto da un set di immagini ed un file csv che contiene le annotazioni per i difetti presenti nelle immagini. Il file CSV contiene 3 colonne, con un elemento per ogni difetto presente nelle immagini, le colonne sono le seguenti:

- **ImageId**: Nome dell'immagine a cui appartiene il difetto.
- **ClassId**: Classe del difetto.
- **EncodedPixels**: Rappresenta la posizione del difetto nell'immagine, come maschera binaria codificata in RLE (Run Length Encoding).

Di seguito alcuni dati estratti dal dataset in formato originale:

| ImageId | ClassId | EncodedPixels |
|---|---|---|
| 0002cc93b.jpg | 1 | 29102 12 29346 24 29602 24 29858 24 30114 24 30370 24 ... |
| 0007a71bf.jpg | 3 | 18661 28 18863 82 19091 110 19347 110 19603 110 ... |

Tabella 4.1: Esempio di dati estratti dal dataset in formato originale.

La maschera codificata in RLE è una sequenza di numeri interi che rappresentano la posizione dei pixel appartenenti al difetto, ogni coppia di numeri rappresentano una riga di pixel nella maschera, il primo numero rappresenta la posizione del primo pixel di una riga mentre il secondo

numero rappresenta la lunghezza della riga di pixel con valore 1.
Tale codifica però considera la posizione di un pixel rispetto all'immagine come se questa fosse un vettore monodimensionale, quindi considerando un vettore di lunghezza $l = w * h$, dove $w$ è la larghezza dell'immagine e $h$ è l'altezza, corrispondenti nel nostro caso a $w = 1600$ e $h = 256$, dunque con un vettore di lunghezza $l = 409600$. Per ogni coppia di numeri nella sequenza che chiamiamo $p1$ e $p2$, per la decodifica sarà sufficiente settare a 1 tutti i pixel compresi tra $p1$ e $p1 + p2$, escluso il pixel in posizione $p1 + p2$, per poi effettuare un *reshape* del vettore in una matrice di dimensioni $w * h$.
Tale processo è effettuato dalla seguente funzione che prende come argomento la stringa contenente la sequenza di numeri e le dimensioni dell'immagine:

```
1  def rle2mask(rle, h, w):
2  """
3      Convert a run length encoding to a mask
4      Args:
5          rle (str): Run length encoding
6          h (int): Height of the mask
7          w (int): Width of the mask
8      Returns:
9          mask (np.array): Mask
10 """
11
12 mask = np.zeros(h*w, dtype=np.uint8)
13 rle = rle.split()
14 starts = np.asarray(rle[0::2], dtype=np.int32)
15 lengths = np.asarray(rle[1::2], dtype=np.int32)
16
17 for i in range(len(starts)):
18     mask[starts[i]-1:((starts[i]-1)+lengths[i])] = 1
19 mask = mask.reshape((w, h)).T
20
21 return mask
```

Figura 4.3: Funzione che effettua la decodifica di una maschera codificata in RLE.

Questa che è mostrata è la funzione principale dell'oggetto che effettua la lettura del dataset originale, il quale funge da iteratore per l'estrazione dell'immagine e delle maschere corrispondenti in formato numpy con un tensore per l'immagine di dimensione $w * h * 3$, e un tensore di dimensioni $n * w * h$ per le maschere, dove $n$ è il numero di classi di difetti presenti nel dataset.

### 4.2.2 Conversione nel formato line json

Il formato utilizzato per manipolare il dataset nel progetto è il formato *line json*. Questo formato permette di codificare le annotazioni come una serie di json suddivisi in righe in uno stesso file, ognuno dei quali contiene i metadati di un'immagine e le annotazioni per i difetti presenti in essa. Questa tipologia di formato consente di avere un buon rapporto tra struttura dei dati

e efficienza in lettura, in quanto i sample codificati come json danno molta libertà per quanto riguarda la struttura dei metadati, mentre l'utilizzo di un singolo file permette un accesso più agevole alle annotazioni rispetto al caso con un file per ogni *sample*, e di poter leggere una singola riga alla volta, senza necessità di caricare tutto il file in memoria.
Questa struttura presenta soltanto il problema di dover conoscere la posizione del carattere di inizio di ogni json file, almeno se si vuole effettuare una lettura casuale degli esempi, ma questo problema è facilmente risolvibile tenendo traccia dei caratteri di inizio riga in un file separato, il quale può essere caricato in memoria prima della lettura del file contenente le annotazioni. Per accedere ad uno specifico elemento basterà scegliere il numero dell'esempio, convertire l'indice dell'esempio con la posizione del carattere di inizio riga corrispondente, e leggere la riga fino al carattere di fine riga, per poi effettuare la conversione da stringa a dict.

Di seguito è possibile vedere un esempio di come sono strutturati i dati nel formato *line json*:

```
1  0
2  2080
3  2115
4  2150
5  ...
```

Figura 4.4: Esempio di file contenente gli indici degli esempi.

```
1  {"img": "0_0.jpg", "polygons": [{"label": "3", "points": [[[1048, 250], ...]]}]}
2  {"img": "1_0.jpg", "polygons": [{"label": "2", "points": [[[917, 60], [913, 62], ...]]}]}
3  {"img": "2_0.jpg", "polygons": []}
4  {"img": "3_0.jpg", "polygons": []}
5  ...
```

Figura 4.5: Esempio di file contenente le annotazioni nel formato *line json*.

Ogni esempio ha la seguente struttura:

- **img**: Nome del file dell'immagine dell'esempio.
- **polygons**: Lista di oggetti che rappresentano le annotazioni per i difetti presenti nell'immagine. ogni oggetto ha la seguente struttura:
  - **label**: Classe del difetto.
  - **points**: Lista di poligoni che rappresentano i difetti presenti nell'immagine, ogni poligono è una lista di punti, dove ogni punto è una coppia di coordinate $y$ e $x$.

Essendo i json, in questo caso, maggiormente composti da numeri, per rendere più efficiente il processo di caricamento è stato scelto di utilizzare il formato binario, utilizzando la libreria *pbjson* la quale consente di serializzare e deserializzare dict in formato json binario, e viceversa. Tale scelta ha permesso di incrementare la velocità di lettura di circa il 30%.

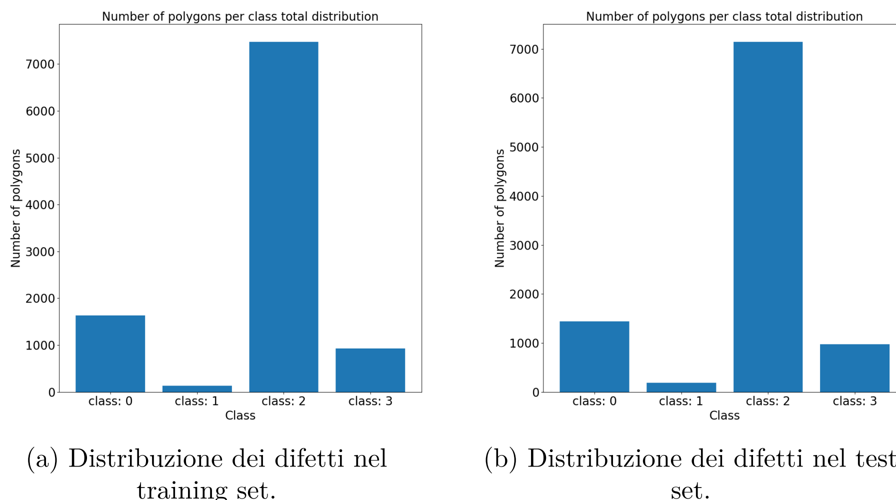
### 4.2.3 Split in train e test set

Per effettuare lo split in train e test set è stato utilizzato un metodo che tiene conto della distribuzione dei difetti nel dataset sorgente e in quelli risultanti dall'operazione di *split*. Tale necessità deriva dal fatto che per la pipeline di addestramento di COIGAN è necessario avere quanti più difetti possibile nel train set, e considerando la distribuzione delle classi, mostrata anche in Tab. 4.2, è evidente che uno splitter casuale non è adatto per questo scopo, rischiando di generare un train o un test completamente privo o quasi di difetti di una certa classe.

Il problema di effettuare lo split del dataset mantenendo la proporzione tra il numero di difetti di una certa classe e il numero di sample del set uguale non è semplicissimo; ogni esempio infatti può avere un diverso numero di oggetti di diverse classi, e dunque tale problema può essere affrontato come un problema di ottimizzazione, dove l'obiettivo è quello di minimizzare la differenza tra il numero di difetti di una certa classe e il numero di difetti target che dovrebbe avere un set per mantenere la medesima distribuzione. Per tale ragione potrebbe essere risolto con un algoritmo di programmazione lineare, ma per questo progetto è stato scelto di utilizzare un algoritmo greedy, che sebbene non garantisca la soluzione ottima, è molto più veloce e semplice da implementare, garantendo comunque un risultato accettabile.

L'algoritmo implementato per lo scopo è presente nel file `COIGAN/training/data/dataset_splitters/fair_splitter.py`, ed utilizza un approccio greedy, calcolando per ogni sample il set con cui ha la maggiore affinità, ovvero viene assegnato al set che ha la maggiore distanza tra il numero di difetti attuale e quello obbiettivo, considerando le variazioni di distanza che si avrebbero assegnando il sample a tale set.

Vediamo di seguito infatti le distribuzioni dei difetti in due set (train e test) generati con questo algoritmo, i quali hanno lo stesso numero di esempi:




(a) Distribuzione dei difetti nel training set.

(b) Distribuzione dei difetti nel test set.

| | Classe 1 | Classe 2 | Classe 3 | Classe 4 |
|---|---|---|---|---|
| **Numero di poligoni train set** | 1637 | 130 | 7475 | 931 |
| **Numero di poligoni test set** | 1445 | 191 | 7147 | 971 |

Tabella 4.2: Distribuzione delle immagini per classe.

### 4.2.4 Creazione del dataset degli oggetti

Una volta generato il training set e il test set, per l'addestramento del modello è necessario creare un dataset separato per ogni classe di difetti i quali contengono esclusivamente i ritagli (crop) dei difetti di quella classe, con le annotazioni relative al crop del difetto. Si ottengono dunque 4 dataset con immagini quadrate di dimensioni arbitrarie per ognuno dei quali è presente una singola annotazione.

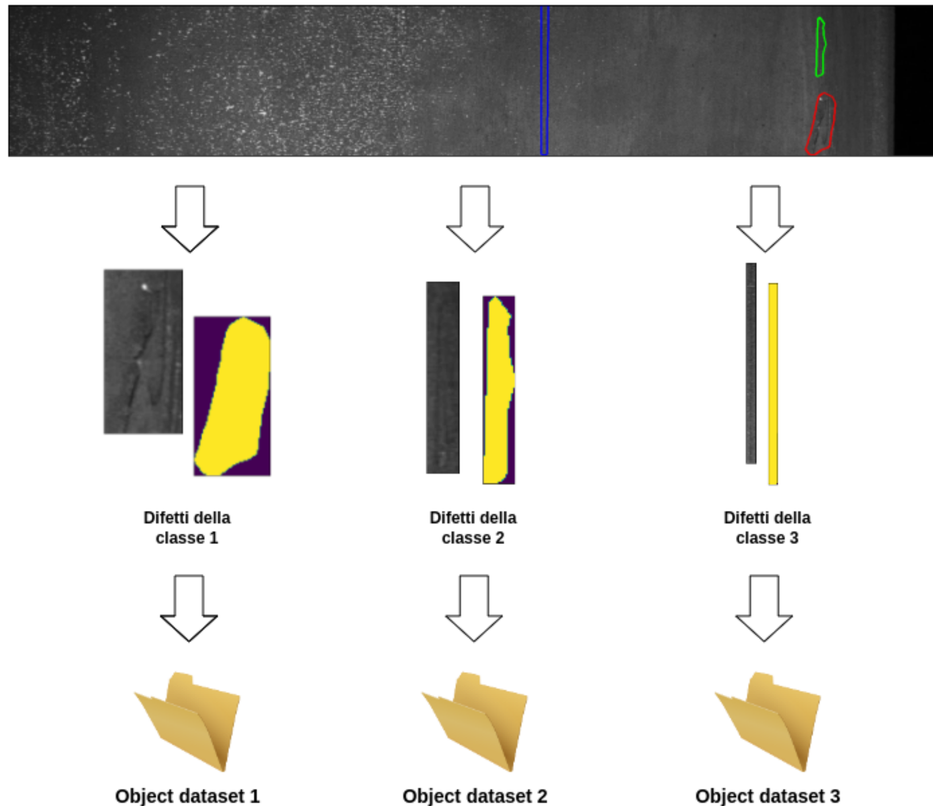



Figura 4.7: Figura che illustra il processo di creazione dei dataset degli oggetti. In questo caso viene mostrato come vengono estratti i difetti da una singola immagine contenente 3 difetti, i quali vengono smistati nei rispettivi dataset.

Per questa tipologia di dataset è stata mantenuto come standard il formato *line json*, ma in quanto in questo caso ad ogni elemento (o immagine) corrisponde una sola annotazione è stata modificata la struttura del json, associando ad ogni elemento una lista di punti di un singolo poligono, ed è stato aggiunto un campo che che indica la dimensione del crop del difetto, tale accortezza permette il caricamento della maschera senza la necessità di caricare l'immagine per caricare le dimensioni della maschera. Di seguito è mostrato un'esempio di come sono strutturati i dati nel formato *line json* per i dataset degli oggetti:

```
1 {"img": "0.jpg", "points": [[[5, 0], [5, 4], ...]], "shape": [246, 22]}
2 {"img": "1.jpg", "points": [[[4, 0], [4, 4], ...]], "shape": [249, 25]}
3 ...
```

Figura 4.8: Esempio di file contenente le annotazioni degli object dataset nel formato *line json*.

### 4.2.5 Creazione del dataset di immagini base

Il dataset di immagini base è utilizzato come base appunto per il processo di inpainting, ed è stato scelto di utilizzare per questo scopo esclusivamente immagini senza difetti, per semplificare la procedura di addestramento e ridurre il numero di considerazioni da fare. Infatti utilizzando tutte le immagini del training set come base ci sarebbe stata la possibilità durante il training di scegliere un'area per la creazione di un difetto che fosse già occupata da un'altro difetto reale con conseguenti disturbi nell'apprendimento.

Un passaggio importante che viene fatto in questa fase, è la procedura di tiling delle immagini, ovvero sono state estratte 7 patch di dimensione 256x256 da ogni immagine, questa scelta è stata guidata principalmente da problematiche legate alla memoria disponibile per la procedura di training che non avrebbe consentito altrimenti di utilizzare una *batch size* di dimensioni adeguate. Di seguito in Figura 4.9 vi è un esempio che illustra graficamente come è stata applicata la procedura.
Il tilling in questo caso è stato effettuato con un leggero overlap tra le patch, per ottenere delle patch quadrate e non sprecare nessuna parte dell'immagine.

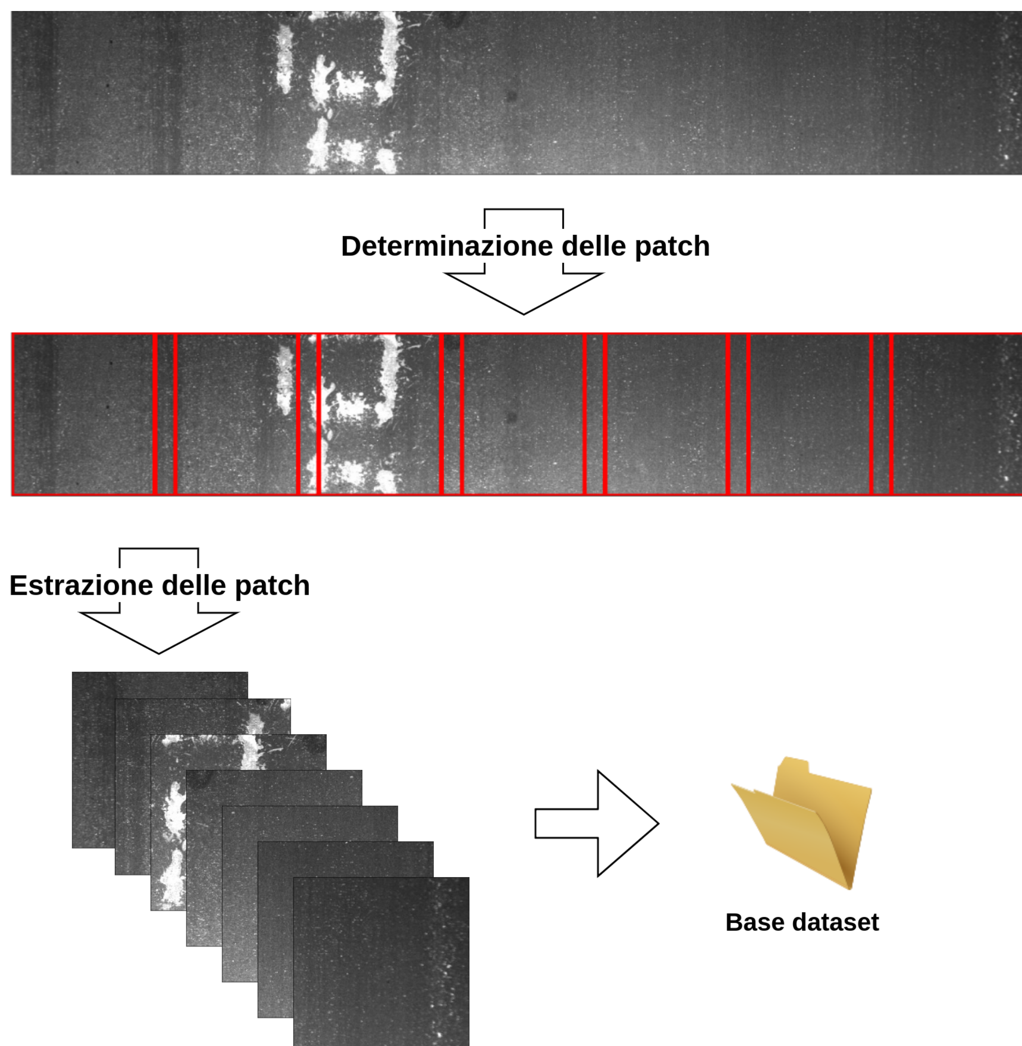



Figura 4.9: Figura che illustra concettualmente il processo di tiling delle immagini.

### 4.2.6 Creazione del dataset utilizzato come riferimento

Il dataset di riferimento segue la stessa procedura effettuata per il dataset di immagini base, con la differenza che in questo caso non sono state selezionate immagini specifiche dal training set, ma sono state semplicemente estratte 7 patch di dimensione 256x256 da ogni immagine. Lo scopo di questo dataset è fornire una distribuzione di riferimento per un discriminatore che confronta le immagini generate con quelle reali, nella loro interezza, principalmente per cercare di attenuare gli *artifacts* generati dal modello sui bordi dei difetti generati.

## 4.3 Il dataloader

Il dataloader in questo progetto ha richiesto più lavoro rispetto ad un modello classico di segmentazione o di generazione di immagini, in quanto doveva gestire 10 sorgenti di dati differenti, le quali poi venivano successivamente combinate, principalmente in tre tensori utilizzati come input per i vari modelli della pipeline.
Data la complessità del dataloader è stato scelto un'approccio modulare, dove ogni componente del dataloader è rappresentata da una classe che a sua volta incapsula o estende altre classi, in modo da rendere più semplice la gestione delle varie operazioni di caricamento dei dati, in Figura 4.10 è mostrato un diagramma UML semplificato del dataloader e delle classi principali che lo compongono, per dare un'idea della struttura dell'oggetto.

Di seguito sono discussi le varie componenti del dataloader, con un approccio bottom-up, partendo dalle classi più periferiche e arrivando a descrivere l'oggetto **CoiganSeverstalSteelDefectsDataset**, che è l'oggetto che viene incapsulato dall'oggetto **torch.utils.data.DataLoader**, utilizzato per effettuare l'addestramento del modello.

### 4.3.1 L'oggetto Augmentor

L'oggetto **Augmentor** è stato realizzato per permettere una gestione agevole delle operazioni di *data augmentation* da applicare a immagini e maschere, in quanto queste necessitano di essere trattate in maniera differente per quanto riguarda alcune specifiche augmentation, mentre per altre necessitano di mantenere una coerenza tra maschera e immagine.

Vediamo due casi che chiariscono la necessità di questa divisione, il primo caso potrebbe essere una rotazione di un angolo arbitrario, applicando separatamente la rotazione ad un'immagine e alla sua maschera, quest'ultima non avrebbe più alcuna coerenza con l'immagine, per tale ragione questo tipo di operazione viene effettuata in maniera sincrona, ovvero applicando la stessa rotazione sia all'immagine che alla maschera.
Per quanto riguarda il secondo caso, possiamo considerare un'operazione di aggiustamento del contrasto, sull'immagine questa operazione può essere applicata senza problemi ma passando la maschera a tale trasformazione si otterrebbe un errore, in quanto la maschera è una matrice

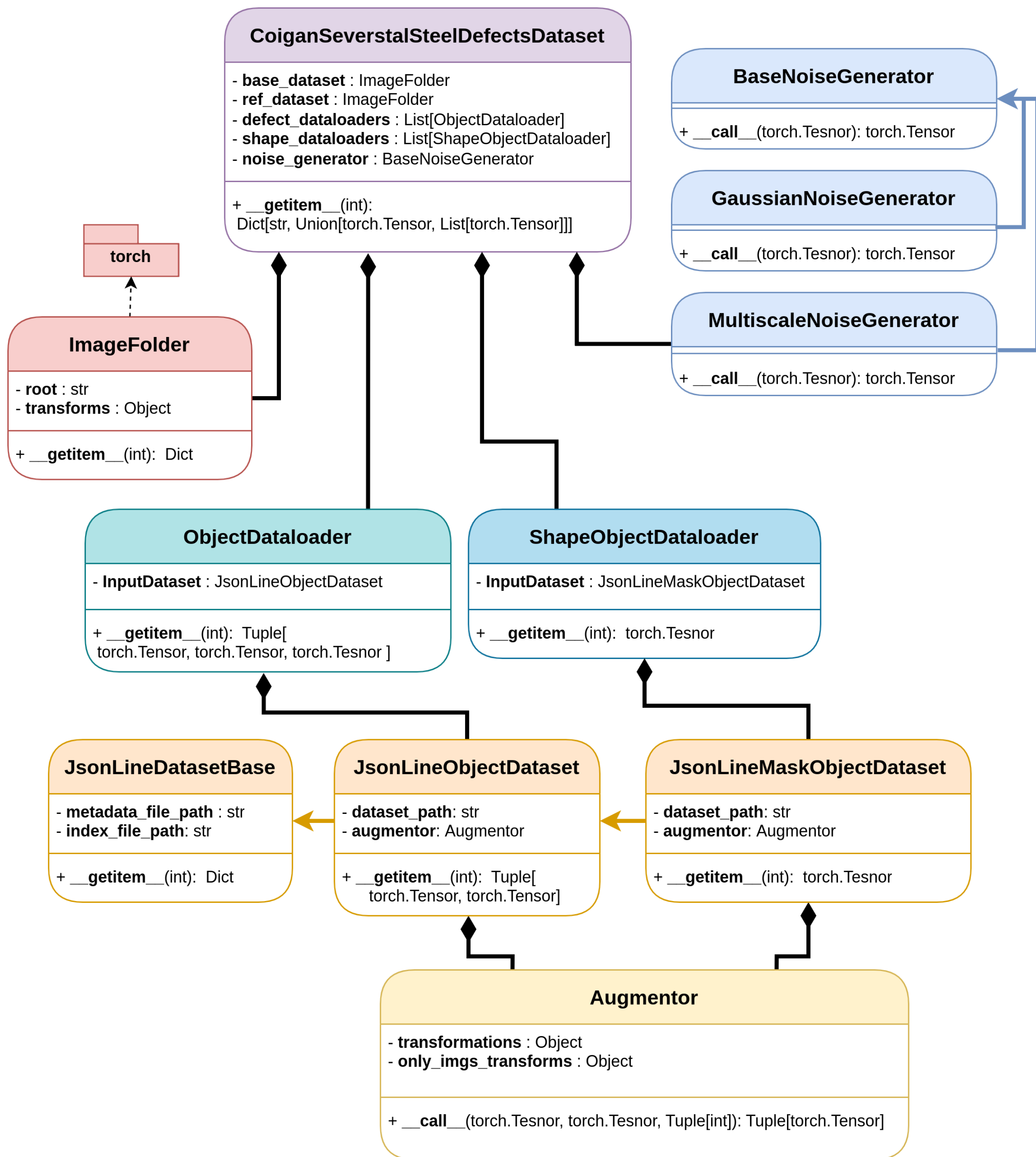



Figura 4.10: In Figura è mostrato il diagramma UML semplificato del dataloader e delle classi principali che lo compongono. Questo infatti suddivide diverse operazioni di caricamento dei dati in classi separate, le quali fanno convergere i risultati all'oggetto **CoiganSeverstalSteelDefectsDataset** che restituisce un output compatibile con l'oggetto **torch.utils.data.DataLoader**.

binaria, con un solo canale, tali considerazioni valgono per ogni altra trasformazione nello spazio dei colori, e in quello della morfologia delle immagini e delle maschere.

Per risolvere tali problematiche sono stati creati degli oggetti che estendono **torch.nn.Module**, e che seguendo la medesima struttura degli oggetti *transformation* presenti nel pacchetto **torchvision.transforms**, applicano le trasformazioni alle immagini e alle maschere in batch, tali metodi sono presenti nel file `COIGAN/training/data/augmentation/custom_transformations.py`.
Come è possibile vedere dal diagramma UML infatti, l'oggetto **Augmentor** accetta due liste di oggetti **torch.nn.Module**, una per le trasformazioni che verranno applicate esclusivamente alle immagini, e una per le trasformazioni che verranno applicate alle maschere e alle immagini.

Una peculiarità di questo oggetto è che consente oltre alle trasformazioni di *data augmentation* anche di applicare delle trasformazioni fisse alla dimensione dei tensori dopo l'augmentation, così da poter adattare le immagini all'input di un modello con dimensione fissa, o semplicemente permettendo di avere tensori della stessa dimensione che possono essere impilati per formare dei batch, in base al caso d'uso.

### 4.3.2 Gli oggetti JsonLineObjectDataset e JsonLineMaskObjectDataset

Il primo layer tra i dati e l'interfaccia finale del dataloader, è rappresentato da due oggetti: **JsonLineObjectDataset** e **JsonLineMaskObjectDataset**. Questi due oggetti che fungono da dataloader intermedi, vengono utilizzati per il caricamento dei dataset dei difetti, i quali sono in formato *line json*, come precedentemente descritto.

Entrambi i dataloader discendono dalla classe **JsonLineDatasetBase**, la quale presenta le primitive per il caricamento dei dataset in formato *line json*, binario e non, senza alcun preconcetto su quale debba essere la struttura degli esempi contenuti al suo interno, dunque potenzialmente utilizzabile per qualunque tipo di dataset basato su questa struttura.

La prima estensione di questa classe è data da **JsonLineObjectDataset**, la quale estende la classe base aggiungendo la funzionalità di caricamento dei dati contenuti negli esempi, ovvero non solo è in grado di caricare i metadati ma anche anche di caricare l'immagine dell'oggetto e la relativa maschera binaria come **torch.Tensor**, in quanto il **JsonLineObjectDataset** si aspetta una determinata struttura all'interno dei sample ottenuti dai metodi della classe madre **JsonLineDatasetBase**. In fine l'oggetto **JsonLineMaskObjectDataset** estende **JsonLineObjectDataset** effettuando un override del metodo **__getitem__**() per restituire esclusivamente la maschera senza effettuare il caricamento dell'immagine, in quanto questo oggetto è utilizzato nei i casi in cui l'immagine RGB non è necessaria.

Entrambi questi oggetti accettano come argomento del costruttore un oggetto **Augmentor** preconfigurato, il quale viene utilizzato per applicare le trasformazioni agli esempi caricati, direttamente durante l'operazione di caricamento, così da avere in uscita da tali oggetti gli esempi con l'*augmentation* già applicata.

### 4.3.3 L'oggetto ObjectDataloader e ShapeObjectDataloader

Un'ulteriore layer di astrazione tra l'uscita del dataloader e i dati sono gli oggetti **ObjectDataloader** e **ShapeObjectDataloader**, questi due infatti vengono caricati nel oggetto **CoiganSeverstalSteelDefectsDataset** e vengono utilizzati per supportare la creazione degli effettivi tensori di uscita. Lo scopo di questi oggetti è quello di caricare i difetti provenienti dai dataset degli oggetti precedentemente menzionati e di posizionarli in dei tensori delle dimensioni adatte alla pipeline di training in maniera casuale, evitando di posizionarli in sovrapposizione ad altri difetti nello stesso tensore.
I due oggetti del dataloader si differenziano principalmente per il fatto che **ObjectDataloader** carica le immagini dei difetti e le loro maschere restituendo un tensore per entrambi, mentre lo **ShapeObjectDataloader** carica solamente le maschere restituendo un unico tensore. Di seguito in figura 4.11 un esempio di come i difetti vengono caricati dal **JsonLineObjectDataset** e poi vengono trasformati dal **ObjectDataloader**.

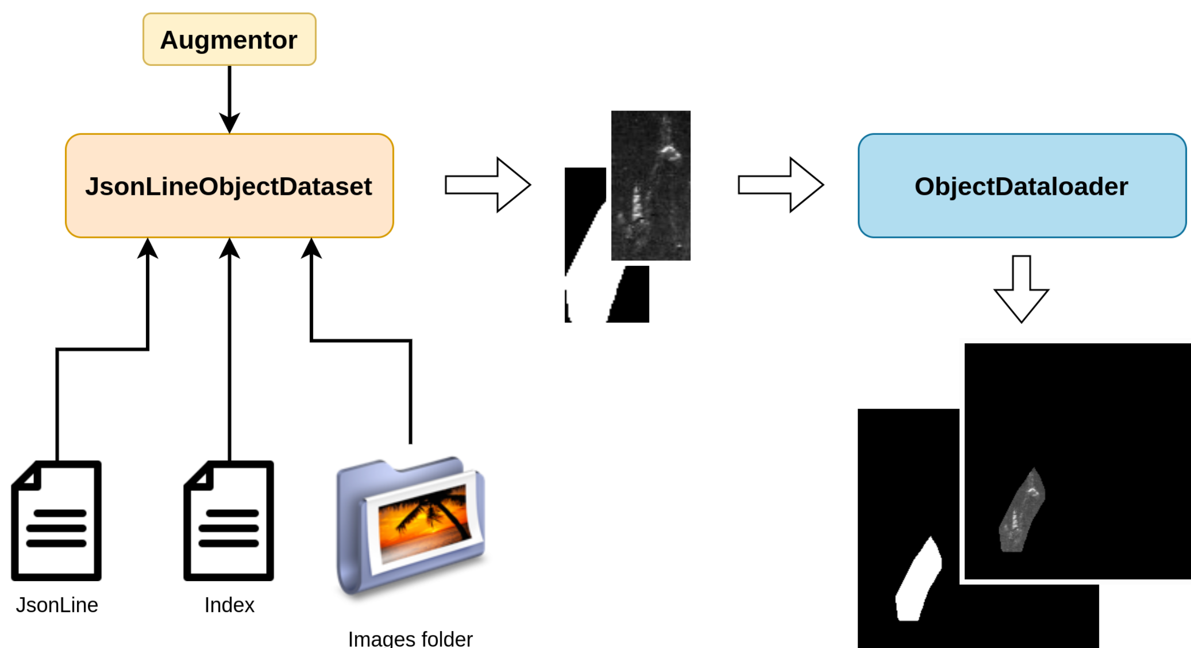



Figura 4.11: Figura che illustra concettualmente il processo di caricamento degli oggetti da parte del **ObjectDataloader**.

### 4.3.4 L'oggetto ImageFolder

La classe ImageFolder è una classe fornita da PyTorch per gestire dataset di immagini in modo semplice ed efficiente. È progettata per lavorare con dataset che seguono una struttura a directory, ed è principalmente utilizzata per dataset di classificazione. La struttura delle directory richiesta da ImageFolder è la seguente:

- root/class_1/sample_1.png
- root/class_1/sample_2.png
- ...

- root/class_2/sample_1.png
- root/class_2/sample_2.png
- ...

Dove root è la directory principale del dataset e class_1, class_2, ecc., sono le sottodirectory che rappresentano le diverse classi di immagini nel dataset. Quando si crea un'istanza della classe ImageFolder, è necessario specificare il percorso alla directory principale del dataset come argomento e il caricamento delle immagini con l'assegnazione delle classi viene effettuato automaticamente. Nel caso di COIGAN è stata utilizzata una sola classe, in quanto l'obbiettivo era semplicemente caricare delle immagini in formato **torch.Tensor**.

### 4.3.5 Il Noise Generator

Il Noise generator è un oggetto che viene utilizzato all'interno di **CoiganSeverstalSteelDefectsDataset** per introdurre del rumore nelle maschere di input del generatore, dove ci si aspetta che il generatore vada a generare dei difetti sull'immagine di input.
L'architettura scelta (vedi 2.5) prevede che il generatore riceva in input un'immagine e delle maschere ma non ha nessun ingresso preferenziale per il rumore, per tale ragione questo viene introdotto direttamente nelle maschere di input, per garantire una buona variabilità dell'output.
Le classi figlie dell'interfaccia **BaseNoiseGenerator**, ovvero: **GassianNoiseGenerator** e **MultiscaleNoiseGenerator**, implementano due metodi di immissione del rumore differenti ma in sintesi, entrambi prendono in ingresso un tensore contenente una maschera binaria e restituiscono un tensore della stessa dimensione con del rumore applicato in corrispondenza dei pixel con valore 1 nella maschera.

Il **GassianNoiseGenerator** è un generatore molto semplice che applica del rumore gaussiano con media e deviazione standard configurabili in corrispondenza dei pixel con valore 1 nella maschera, come mostrato in Figura 4.12.

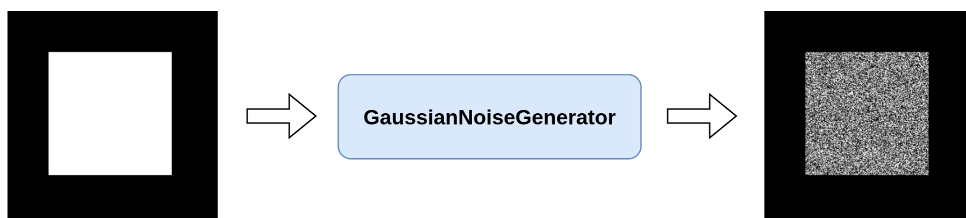



Figura 4.12: Figura che illustra un esempio di applicazione del **GassianNoiseGenerator**.

Il **MultiscaleNoiseGenerator** invece è un generatore più articolato il quale può ospitare al suo interno un'ulteriore generatore di rumore, ed è in grado di applicare il rumore proveniente dal generatore interno a diverse scale per dare una variabilità più realistica al rumore applicato. Per ottenere tale effetto date $n$ scale, ad esempio [1, 8, 16], dato un tensore di ingresso di dimensioni $h\ w$, il generatore crea 3 tensori di rumore con rispettivamente dimensioni $h\ w$, $\frac{h}{8}\ \frac{w}{8}$ e $\frac{h}{16}\ \frac{w}{16}$, e riporta tramite interpolazione bilineare tutti i tensori alla dimensione $h\ w$, per poi sommarli, come è illustrato nella figura 4.13.

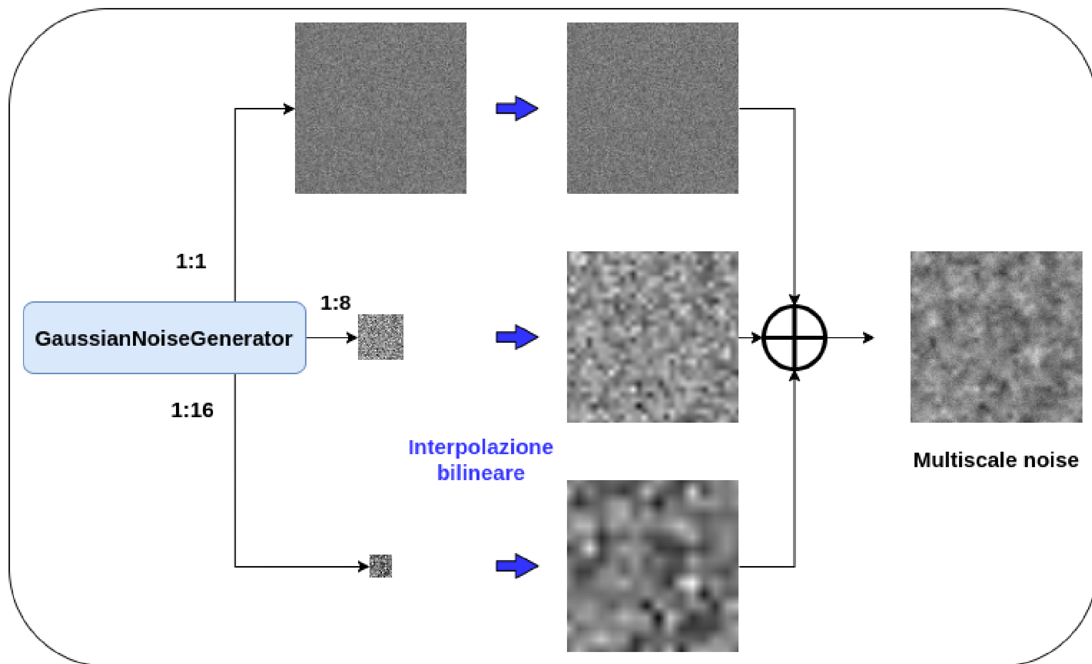



Figura 4.13: Figura che illustra un esempio di generazione di rumore all'interno del **MultiscaleNoiseGenerator**.

Per mitigare l'effetto di discontinuità tra le aree dell'output del modello in cui viene richiesta la generazione di un difetto e le aree circostanti, le quali tendevano a generare degli artifacts durante i primi addestramenti è stata introdotta un'ulteriore *feature* al **MultiscaleNoiseGenerator**, ovvero la possibilità di applicare un smooth gaussiano alla maschera prima di applicare il rumore. Tale tecnica consente una attenuazione graduale del rumore allontanandosi dal difetto, contribuendo a mitigare gli artifacts in tali aree. Nella figura 4.14 è possibile vedere un esempio di come viene applicato lo smoothing e il rumore generato nel passaggio precedente alla maschera in ingresso.

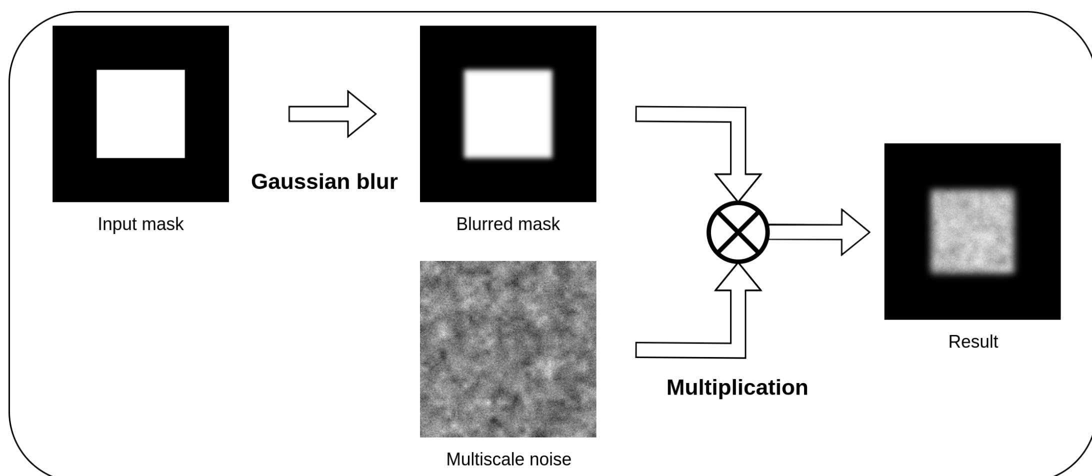



Figura 4.14: Figura che illustra un esempio di applicazione del rumore generato dal **MultiscaleNoiseGenerator**.

### 4.3.6 L'oggetto CoiganSeverstalSteelDefectsDataset

L'oggetto della classe **CoiganSeverstalSteelDefectsDataset** dunque è un contenitore che si occupa di caricare tutti i dati pre-elaborati dai vari dataloader concatenandoli dove necessario, applicando in oltre alcune modifiche finali ai tensori da passare alla pipeline di addestramento.

Abbiamo visto dunque dallo schema UML e per quanto detto precedentemente che questo oggetto raggruppa 6 dataloader differenti, ovvero 2 **ImageFolder**, uno per il dataset delle immagini base e uno per le immagini da riferimento per uno dei discriminatori, mentre ha ben 4 dataloader differenti per i dataset degli oggetti, uno per ogni classe, i quali come visto caricano le immagini e le maschere degli oggetti, tali difetti vengono poi utilizzati per addestrare il discriminatore dei difetti, mentre le maschere sono utilizzate anche come input per il generatore. Tale configurazione è mostrata nella figura 4.15.

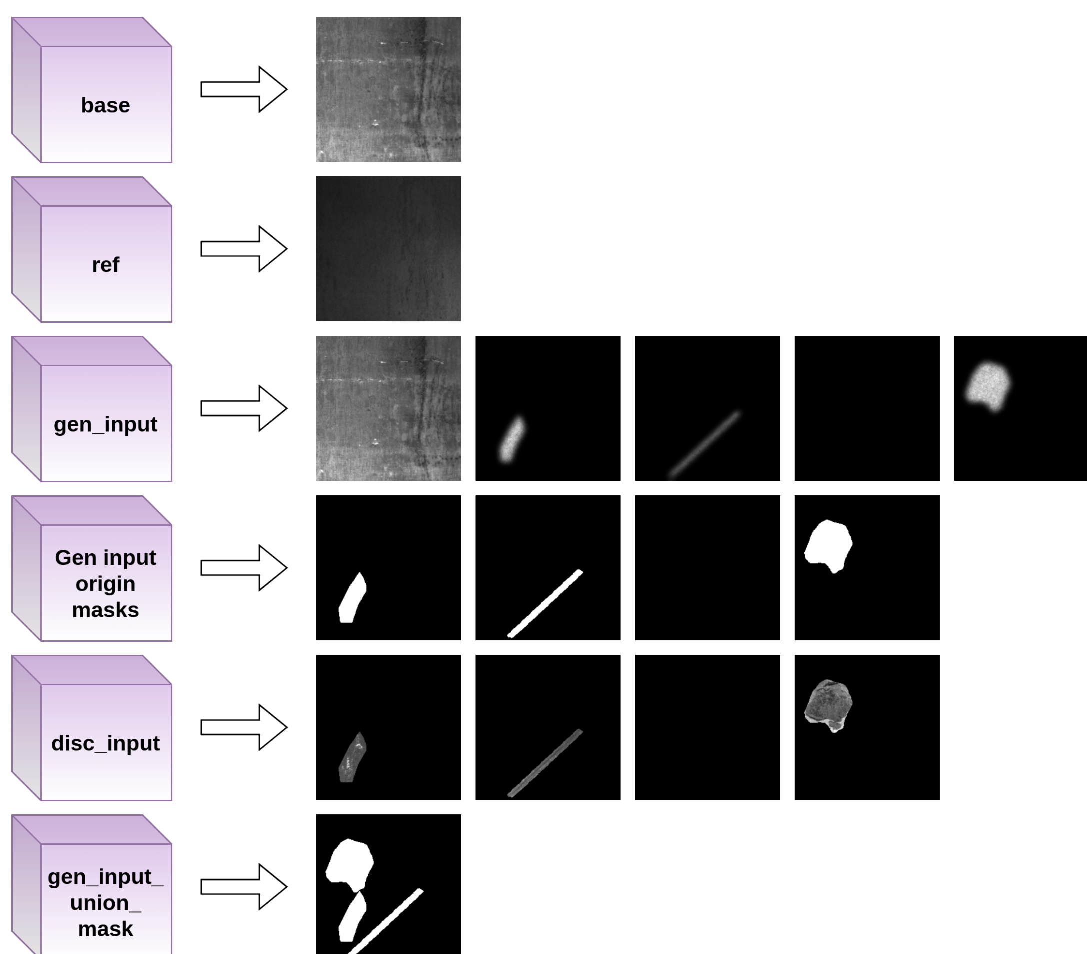



Figura 4.15: Nella figura è mostrato il contenuto di un sample che si può ottenere dall'oggetto **CoiganSeverstalSteelDefectsDataset**, a sinistra i cubi rappresentano i tensori, con i nomi effettivi che sono presenti nel dict, mentre a destra vi sono i canali dei tensori, le immagini raggruppano 3 canali mentre le maschere occupano un solo canale. Si noti che l'ordine da sinistra verso destra è coerente con l'ordine dei canali nei tensori.

Con un'impostazione però è possibile richiedere maschere di input diverse dai difetti passati al discriminatore, arrivando a 10 dataloader differenti, in quanto si aggiungono ulteriori 4 dataloader per la generazione delle maschere di input, le quali verranno generate da dei

dataloader **ShapeObjectDataloader** che caricano esclusivamente le maschere degli oggetti e non le immagini, anche se effettuano la lettura degli stessi dataset.

L'oggetto restituito dal **CoiganSeverstalSteelDefectsDataset** come sample è un *dict* che associa delle stringhe a dei tensori, tali tensori sono illustrati in figura 4.15 dove i nomi sui cubi che rappresentano i tensori sono gli effettivi nomi utilizzati nel dict. Verrà poi illustrato come questi tensori vengono utilizzati nella pipeline di addestramento nella prossima sezione.

In sintesi vediamo le operazioni svolte dall'oggetto della classe **CoiganSeverstalSteelDefectsDataset**:

- Estrae un'immagine base dal dataset delle immagini base.
- Estrae un'immagine di riferimento dal dataset di riferimento.
- Estrae delle immagini e delle maschere dei difetti dall'**ObjectDataloader** per ogni classe.
- Estrae delle maschere dei difetti dallo ShapeObjectDataloader per ogni classe, se specificato nelle configurazioni altrimenti riutilizza le maschere caricate in precedenza.
- Se richiesto effettua il masking della base image, con le maschere dei difetti che verranno passate in input al generatore.
- Applica il rumore alle maschere di input del generatore.
- Effettua la *stack* dei tensori per creare i tensori di input per il generatore e i discriminatori.
- Ritorna i tensori come dict.

Il *dict* restituito da questo oggetto viene poi processato dal **torch.utils.data.DataLoader**, il quale oltre a gestire il caricamento dei dati, si occupa anche di creare i batch, e di gestire il caricamento tramite istanze multiple di questo oggetto utilizzando il *multiprocessing* di Python.

## 4.4 La pipeline di addestramento

In questa sezione verranno illustrate nel dettaglio le componenti della pipeline di addestramento, mostrando effettivamente come vengono utilizzati i dati caricati dai dataloader precedentemente mostrati, verrà mostrato come vengono calcolate e utilizzate le loss, e le regolarizzazioni per l'addestramento dei modelli.

### 4.4.1 Il generatore

Il generatore è la componente principale della pipeline, e come già detto utilizza la medesima architettura di LaMa, l'unica modifica che è stata apportata alla struttura del modello è la dimensione del tensore di ingresso il quale può variare in base al numero di classi di oggetti che si vogliono generare.
La loss di questo modello è composta da 4 elementi come mostrato nella figura 4.2 più la

regolarizzazione, ovvero: le non saturating loss calcolate con l'output dei discriminatori, la perceptual loss smooth masked, la L1 smooth masked e in fine la *path-length regularization*.
In questa parte andiamo ad analizzare in dettaglio come sono state implementate la *non saturating loss*, la *path-length regularization* e la L1 loss, mentre la perceptual loss varrà discussa in seguito.

### La non saturating loss

Nella loss totale del generatore sono presenti due componenti relative al discriminatore dei difetti e al discriminatore delle immagini, per i quali si utilizza la *non saturating loss* come nella paper originale di Goodfellow [11], la quale nel codice è definita come segue:

```
1   def g_nonsaturating_loss(discriminator_output):
2       return loss = F.softplus(-discriminator_output).mean()
```

Tale formulazione della *non saturating loss* è comprensiva della funzione di attivazione sigmoide, in quanto i modelli dei discriminatori non ne sono provvisti, infatti abbiamo:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

$$softplus(x) = \log(1 + e^{x}) \tag{4.2}$$

$$L_G(D) = \mathbb{E}_{z \sim p_z(z)}[-\log(\sigma(D(G(z))))] = \mathbb{E}_{z \sim p_z(z)}[softplus(-D(G(z)))] \tag{4.3}$$

é possibile notare come la concatenazione della sigmoide 4.1 e dalla *non saturating loss* sia equivalente alla funzione *softplus* 4.2 applicata all'output del discriminatore, come mostrato in 4.3.

### La loss L1 smooth masked

Per quanto riguarda la L1 loss, come accennato all'inizio di questo capitolo, è stata implementata una versione modificata, chiamata **L1 smooth masked**, questa loss viene applicata per calcolare la differenza tra l'immagine generata e l'immagine base utilizzata per l'inpainting, pesando in maniera differente le aree dove è stato effettuato l'inpainting e le aree dove l'immagine deve combaciare con l'immagine base.
Rispetto all'implementazione originale presente in LaMa è stata aggiunta una convoluzione della maschera utilizzata per il calcolo dei pesi delle varie aree, tale tecnica è stata utilizzata per mitigare la formazione di artifacts nelle aree di transizione tra le aree in cui è stato effettuato l'inpainting e non.
Vediamo di seguito il metodo che applica la loss L1 smooth masked:

```
def __call__(self, pred, target, mask):
    """
    Args:
        pred (torch.Tensor): prediction tensor
        target (torch.Tensor): target tensor, reference for the prediction
        mask (torch.Tensor): mask tensor, with values in {0, 1},
            where 1 means the pixel correspond to an impainted area.
    """

    # create the smoothed mask
    if self.kernel_size > 0:
        mask = F.conv2d(mask, self.kernel, padding=self.kernel_size // 2)

    # convert the mask in a weight mask
    weight_mask = self.obj_weight * mask + self.bg_weight * (1 - mask)

    # apply the masked L1 loss
    loss = F.l1_loss(pred, target, reduction='none')
    loss = loss * weight_mask

    return loss.sum() / weight_mask.sum()
```

Quello mostrato è il metodo ___call___ della classe **SmoothMaskedL1** presente nel file `COIGAN/training/losses/masked_losses/smooth_masked_l1.py`, il quale effettua i seguenti passaggi:

- **Riga 13:** Effettua lo smoothing della maschera in ingresso, che indica le aree in cui è stato effettuato l'inpainting, utilizzando una convoluzione 2D.
- **Riga 16:** Effettua la conversione della maschera in un tensore di pesi, dove i pixel con valore 1 nella maschera vengono moltiplicati per il peso assegnato alle aree in cui è stato effettuato l'inpainting, mentre la maschera invertita viene moltiplicata per il peso assegnato al resto dell'immagine.
- **Riga 19:** Applica la L1 loss tra l'immagine generata e l'immagine base, senza effettuare la riduzione, in modo da ottenere un tensore di loss con la stessa dimensione dell'immagine.
- **Riga 20:** Moltiplica il tensore della loss per il tensore dei pesi, in modo da pesare in maniera differente la loss associata alle diverse aree.
- **Riga 22:** Effettua la riduzione della loss, dividendo la somma del tensore di loss per la somma del tensore di pesi, in modo da ottenere una loss normalizzata rispetto ai pesi.

Vediamo di seguito la figura 4.16 la quale illustra graficamente i passaggi della loss L1 smooth masked, applicati ad un'immagine esempio alla quale simuliamo di applicare l'inpainting dove è specificato da una maschera binaria, per simulare il processo è stato applicato del rumore gaussiano su tale punto, ed è si è utilizzato un peso di 1.0 per il background mentre si è utilizzato

un peso di 0.0 per l'area indicata dalla maschera, e si è utilizzato un kernel di dimensione 5x5 per lo smoothing della maschera.

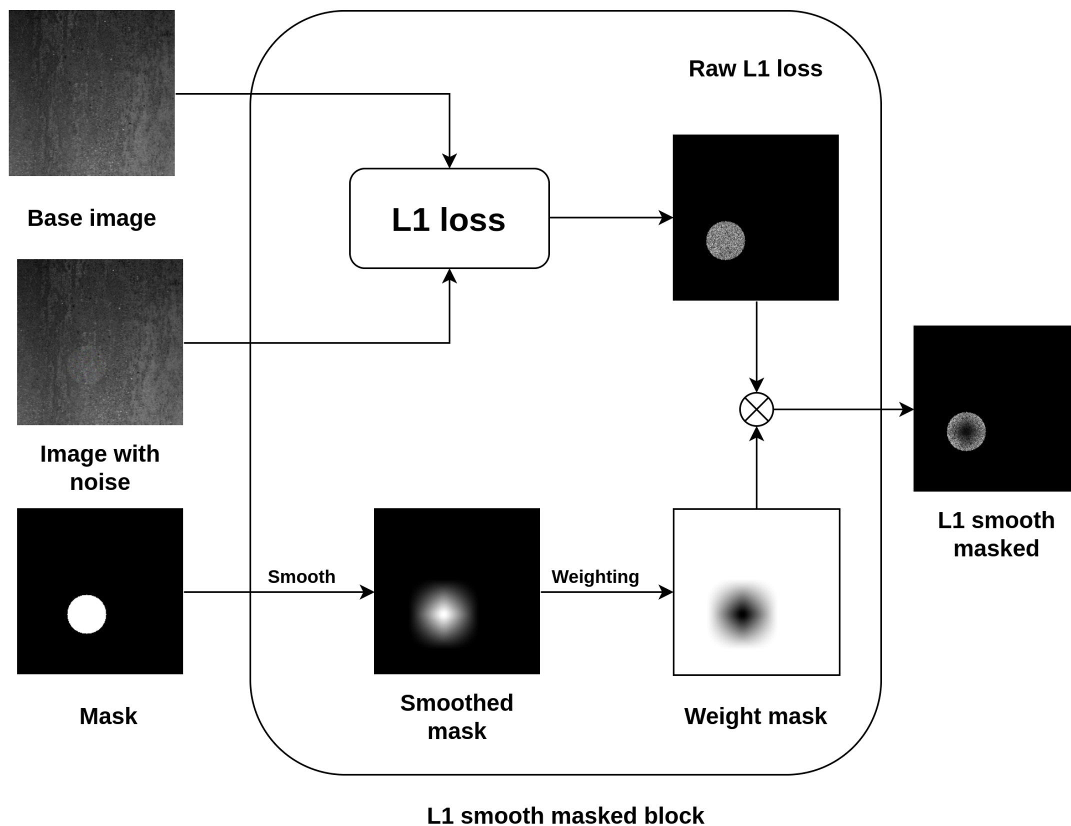



Figura 4.16: Figura che illustra graficamente il processo di calcolo della loss L1 smooth masked.

Il risultato di questa operazione è che anche un'area localizzata intorno al bordo dell'area generata sarà soggetta ad una penalizzazione, dovendo restare simile all'immagine base, con un peso che scende gradualmente dall'esterno verso l'interno della maschera, in tal modo non vi è più una transizione netta, in tal modo si spinge il modello a generare un difetto che abbia una maggiore coerenza con l'immagine base, e penalizza maggiormente la creazione di bordi netti tra le due aree e la generazione di artifacts. Si noti che per l'esempio il rumore è stato generato con la maschera, dunque il calcolo della loss senza lo smoothing della maschera utilizzata per calcolare la matrice dei pesi avrebbe ottenuto una loss nulla, in quanto le aree con un valore della *Raw L1 loss* sarebbero state associate a pesi nulli.

### La path-length regularization

La *path-length regularization* è una tecnica utilizzata per stabilizzare il training del generatore, la quale è stata introdotta nella paper di Karras et al. [17] per il modello stylegan.
L'obbiettivo di questa tecnica di regolarizzazione è ottenere una maggiore stabilità del modello durante la fase di training, penalizzando grandi variazioni del tensore di output causate da piccole variazioni del tensore di input.

Per misurare queste variazioni si considera il gradiente del tensore di output rispetto al tensore di input, e si calcola la variazione del gradiente al variare della direzione in cui ci si sposta

nello spazio dell'input, per variazioni di intensità simile in diverse direzioni si dovrebbe dunque avere delle variazioni nell'output simili, tale condizione indica che lo spazio dell'input è ben strutturato.

Considerando $x \in \mathbb{X}$ lo spazio dei tensori in ingresso e $y \in \mathbb{Y}$ lo spazio dei tensori in uscita di $G$, definiamo $G$ come una funzione $G(x) \rightarrow y$, e definendo un punto $x_0 \in \mathbb{X}$, possiamo definire le proprietà locali del generatore nell'introno di $x_0$ con la matrice Jacobiana:

$$J_G(x_0) = \frac{\partial G(x_0)}{\partial x_0} \tag{4.4}$$

E si definisce dunque la *path-length regularization* come:

$$\mathbb{E}_{x_0,y}(||J_G(x_0)^T \cdot y||_2 - a)^2 \tag{4.5}$$

Dove abbaiamo che $a$ rappresenta una costante dinamicamente definita, attraverso la media mobile esponenziale delle lunghezze $||J_G(x_0)^T \cdot y||_2$ calcolate durante le iterazioni precedenti. Un'ulteriore punto importante dell'implementazione della *path-length regularization* è lo sfruttamento dell'identità $J_G(x_0)^T \cdot y = \nabla_{x_0}(G(x_0) \cdot y)$, la quale consente di calcolare la matrice Jacobiana in modo efficiente attraverso il backpropagation.

La regolarizzazione nel caso di questo progetto è stata applicata ogni 4 step di addestramento del generatore, moltiplicandone per 4 il valore della loss, in quanto si è osservato che il risultato rimane simile, mantenendo comunque stabile l'addestramento del generatore, riducendo però il carico computazionale della procedura durante l'addestramento.

### 4.4.2 La perceptual loss smooth masked

La perceptual loss è stata utilizzata con le medesime modalità descritte nella sezione 2.5.2, le uniche differenze di implementazione, sono legate all'utilizzo di una maschera per pesare in maniera differente le aree dell'immagine, come già mostrato per la **L1 smooth masked**, in questo caso però invece di applicare la moltiplicazione ad un solo tensore, è necessario scalare la maschera dei pesi per le dimensioni di ognuna delle features di ogni layer del modello utilizzato per il calcolo della loss, prima di applicarla ed effettuare la riduzione.

Di seguito la funzione *forward* della classe **ResNetPLSmoothMasked** presente nel file `COIGAN/training/losses/masked_losses/smooth_masked_resnet_perceptual.py`, nella quale è possibile vedere come viene effettuata l'inferenza di un modello ResNet50, il quale restituisce una lista di tutte le features interne, e come vengono create delle copie della maschera dei pesi scalate attraverso interpolazione bilineare, per coincidere con le dimensioni dei tensori dei vari layer.

Non verrà illustrato graficamente a causa della elevata dimensionalità dei tensori e in quanto il meccanismo è analogo a quello della **L1 smooth masked**.

```
def forward(self, pred, target, input_mask):
    """
    Compute the ResNet perceptual loss for the input and target.
    Args:
        pred: predicted tensor
        target: target tensor
        input_mask: input mask tensor
    Returns:
        ResNet perceptual loss
    """
    pred = (pred - IMAGENET_MEAN.to(pred)) / IMAGENET_STD.to(pred)
    target = (target - IMAGENET_MEAN.to(target)) / IMAGENET_STD.to(target)
    pred_feats = self.impl(pred, return_feature_maps=True)
    target_feats = self.impl(target, return_feature_maps=True)

    # Compute the mask
    if self.kernel_size > 0:
        mask = F.conv2d(input_mask, self.kernel, padding=self.kernel_size // 2)

    # convert the mask in a weight mask
    weight_mask = self.obj_weight * mask + self.bg_weight * (1 - mask)

    # create a weight mask for each feature layer
    resized_weight_masks = [F.interpolate(weight_mask, size=feat.shape[2:], \
            mode=self.interpolation_mode, align_corners=self.allign_corners)
        for feat in pred_feats]

    # Compute the loss
    layer_losses = []
    for cur_pred, cur_target, w_mask in zip(pred_feats, target_feats, resized_weight_masks):
        layer_loss = F.mse_loss(cur_pred, cur_target, reduction='none')
        masked_layer_loss = layer_loss.sum(dim=1, keepdim=True) * w_mask
        layer_losses.append(masked_layer_loss.sum() / w_mask.sum())
    return torch.stack(layer_losses).mean()
```

Nel codice presentato sono effettuati i seguenti passaggi:

- **Riga 11-12:** Effettua la normalizzazione dei tensori di input e target, utilizzando i valori di media e deviazione standard del dataset *ImageNet*.
- **Riga 13-14:** Estrae le features interne del modello ResNet50 per i due tensori di input.
- **Riga 18:** Effettua lo smoothing della maschera di input, utilizzando una convoluzione 2D.
- **Riga 21:** Effettua la conversione della maschera in un tensore di pesi della loss.
- **Riga 24-26:** Effettua la generazione delle copie della maschera dei pesi scalate per le dimensioni dei tensori delle features interne del modello ResNet50.
- **Riga 29-34:** Calcola la loss per ogni layer, e applica la maschera dei pesi, per poi ottenere la loss finale come media delle distanze pesate di ogni layer.

### 4.4.3 Il discriminatore dei difetti

Il discriminatore dei difetti è il modello utilizzato per il calcolo della loss principale, la quale ha il compito di portare il generatore a generare dei difetti con una distribuzione quanto più simile a quella dei difetti provenienti dal dataset di training. Il modello utilizzato per il training è un discriminatore convoluzionale, con la struttura proposta da Gal et al. [8], in tale pubblicazione è stata proposta una architettura basata sulla trasformata *wavelet*, la quale sembra ottenere risultati migliori per quanto riguarda dettagli in alta frequenza delle immagini.

Di seguito in figura 4.17 è rappresentato uno schema concettuale che illustra la struttura interna del discriminatore basato sulla trasformata *wavelet*. In tale schema i blocchi DWT e IWT rappresentano rispettivamente la trasformata *wavelet* discreta e la sua inversa, mentre i blocchi *Down* effettuano un *downsampling* dell'immagine attraverso una interpolazione bilineare nel dominio spaziale, per poi riportare il tensore nel dominio *wavelet* attraverso la trasformata.

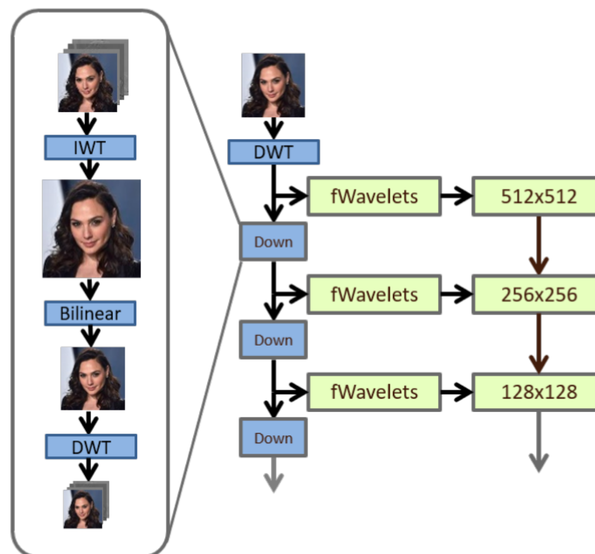



Figura 4.17: Figura che illustra concettualmente la struttura interna del discriminatore basato sulla trasformata *wavelet*. credits: Rinon Gal et. al [8]

Una parte importante dell'addestramento di questo discriminatore, risiede nel preprocessamento dell'input; infatti il modello non riceve in input l'immagine generata così come è ma, riceve esclusivamente le aree relative ai difetti. Oltre all'operazione di *masking* è stata applicata una separazione dei difetti in base alla classe ottenendo così un'immagine per ogni classe, le quali vengono poi messe in *stack* per ottenere un tensore di dimensione $(n \cdot c) \times h \times w$, dove $n$ è il numero di classi, $c$ è il numero di canali di ogni immagine, $h$ e $w$ sono rispettivamente l'altezza e la larghezza dell'immagine. Tale tensore è poi passato in input al discriminatore, il quale restituisce un singolo valore che indica se l'immagine in input è stata generata o appartiene alla distribuzione reale.
Di seguito è rappresentata l'operazione di estrazione dei difetti dall'immagine generata. I blocchi con sfondo arancione rappresentano dei tensori, i quali contengono in *stack* le immagini al loro interno, in particolare abbiamo nel primo blocco l'immagine base con le maschere associate alla posizione dove andranno generati i difetti, mentre nel secondo blocco è presente l'immagine generata, e nel terzo blocco sono presenti le immagini dei difetti estratti dall'immagine generata.

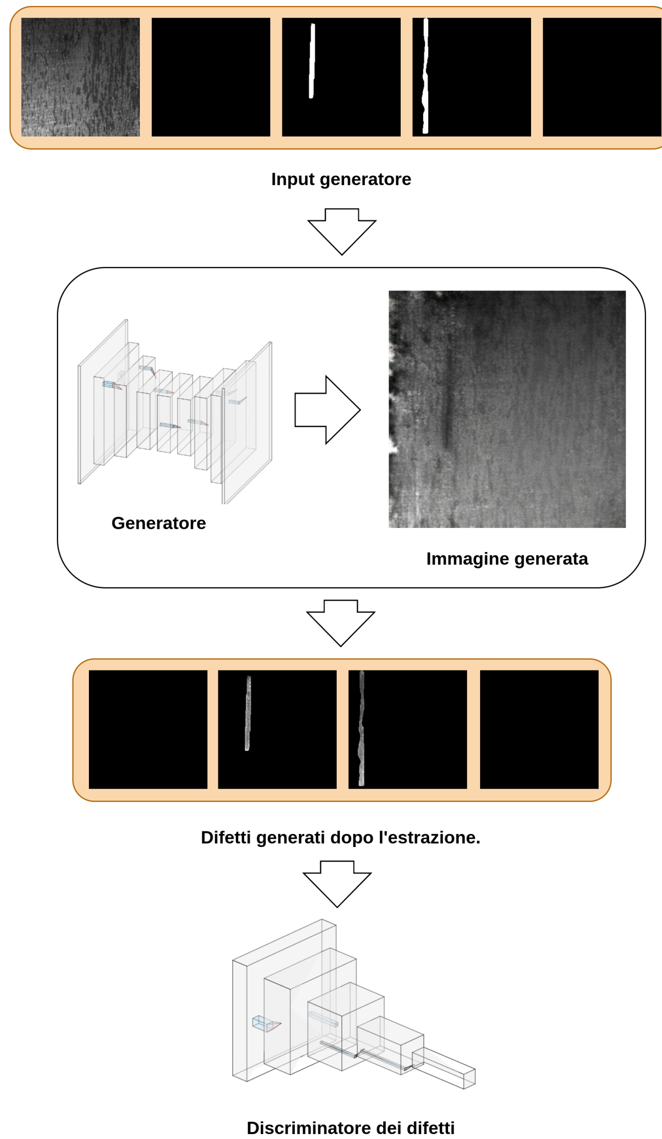



Figura 4.18: Figura che illustra concettualmente il processo di estrazione dei difetti dall'immagine generata, per l'addestramento del discriminatore.

Per l'addestramento del discriminatore è stata utilizzata la loss function proposta nella paper originale di Goodfellow et al. [11]:

$$L_D(G) = \mathbb{E}_{x \sim p_{data}(x)}[-\log(D(x))] + \mathbb{E}_{x \sim p_{data}(x)}[-\log(1 - D(G(x)))] \tag{4.6}$$

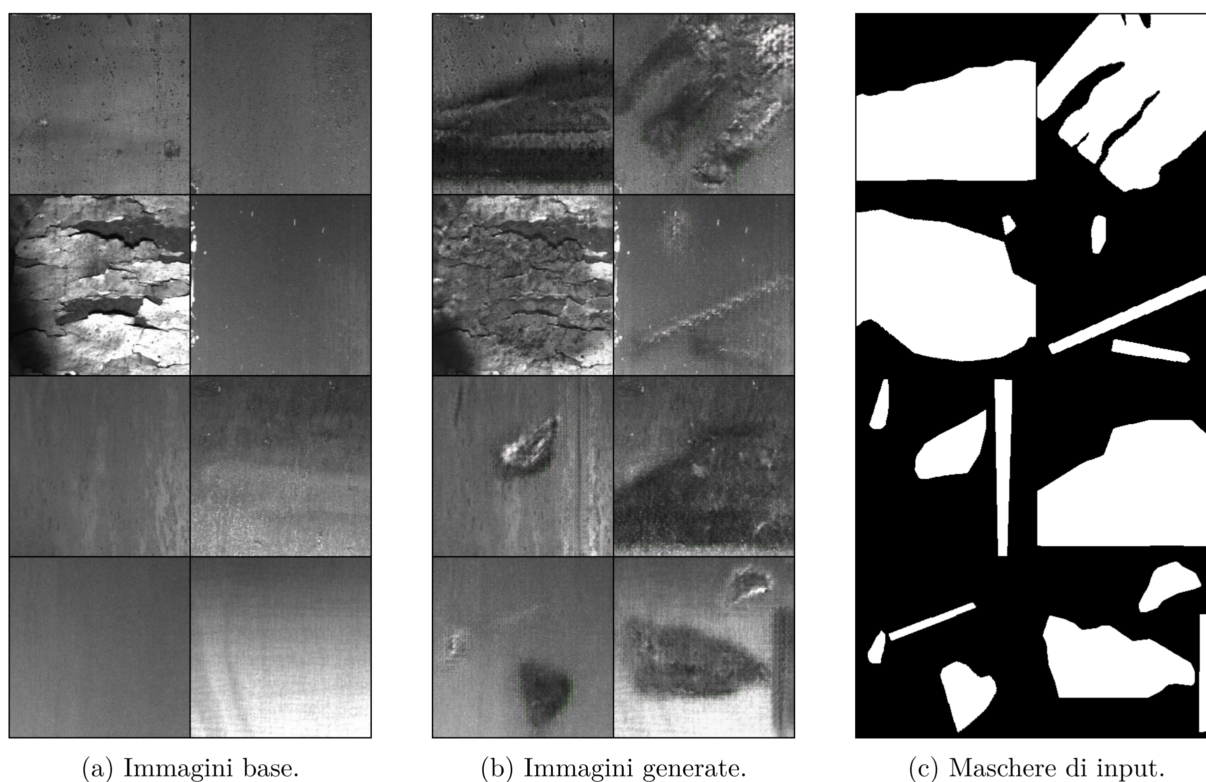
Allo stesso modo anche la adversarial loss per il generatore ottenuta da questo discriminatore è quella originale proposta nella medesima pubblicazione:

$$L_G(D) = \mathbb{E}_{x \sim p_{data}(x)}[-\log(D(G(x)))] \tag{4.7}$$

Per questo discriminatore come metodo di regolarizzazione è stata utilizzata la R1, illustrata estensivamente nella sottosezione 2.5.2.

### 4.4.4 Il discriminatore di riferimento

Il discriminatore di riferimento è il modello utilizzato per il calcolo di una loss supplementare, la quale è stata introdotta per mitigare il problema degli artifacts sui bordi delle aree in cui è stato effettuato l'inpainting, e per cercare di risolvere un'ulteriore problema di stacco delle tonalità di colore tra le aree rigenerate e quelle che dovrebbero essere mantenute coerenti con l'immagine base. Nella figura 4.19 sono mostrati alcuni esempi di immagini generate affette da artifacts e stacco delle tonalità di colore:

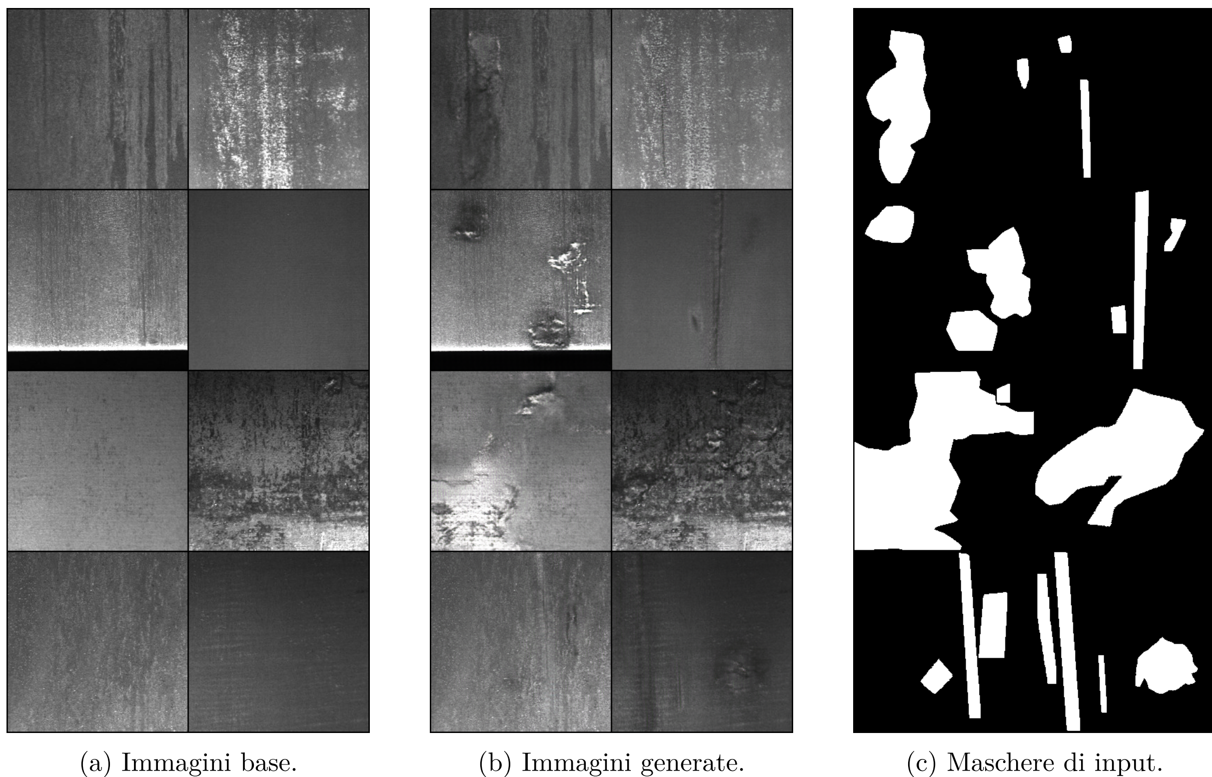


(a) Immagini base. (b) Immagini generate. (c) Maschere di input.

Figura 4.19: Figura che illustra un batch di immagini generate, da una precedente versione della training pipeline, non provvista di **ref discriminator**, e con le loss **L1** e **Perceptual loss** che non utilizzano lo smooth della weight mask.

Nella figura 4.20 invece è mostrato un batch preso da un training con una versione della training pipeline che utilizza il **ref discriminator**, e con le loss **L1** e **Perceptual loss** che applica lo smooth della weight mask ed è possibile vedere come il risultato sia molto migliorato. Il discriminatore aggiuntivo ha introdotto però un comportamento inatteso del modello, il quale tende ora ad alterare maggiormente anche l'area dell'immagine che non è interessata dalle maschere per la generazione dei difetti, anche se generalmente non genera difetti dove non richiesto e mediamente la qualità dei risultati ottenuti è maggiore.

Anche per il discriminatore di riferimento sono state utilizzate le stesse loss e regolarizzazioni utilizzate per il discriminatore dei difetti.



(a) Immagini base. (b) Immagini generate. (c) Maschere di input.

Figura 4.20: Figura che illustra un batch di immagini generate, da una versione della training pipeline provvista di **ref discriminator**, e con le loss **L1** e **Perceptual loss** che utilizzano lo smooth della weight mask.

## 4.5 La valutazione del modello

La valutazione dei risultati ottenuti da un modello di inpainting è un'operazione complessa, la quale teoricamente non può essere effettuata in maniera accurata, in quanto per valutare la qualità di un oggetto generato non abbiamo un effettivo esempio di riferimento dalla quale calcolare la distanza in maniera precisa.

Un primo tentativo di risolvere questo problema è stato proposto da parte di Tim Salimans et al. [26] nel 2016, i quali hanno proposto un metodo chiamato **Inception score**, il quale utilizza un modello di *image classification*, in particolare un modello **InceptionV3** pre-addestrato, per effettuare l'inferenza su un elevato numero di immagini, (sono state utilizzate 50000 immagini generate dal modello GAN), ed utilizzare tali valori come base per stabilire la qualità del modello. Il metodo in particolare consisteva nel valutare l'entropia delle distribuzioni delle classi predette dal modello di classificazione $p(y|x)$. Tale metodo ha mostrato una certa somiglianza con i risultati ottenuti dagli annotatori umani attraverso lo strumento di **AWS Mechanical Turk**.

L'approccio di valutazione con annotatori umani ha sottolineato maggiormente la difficoltà di valutare tale tipo di risultati, questi infatti dovevano valutare se un'immagine presentata fosse reale o generata assegnando una valutazione da 1 a 5. Durante il processo venivano casualmente presentate immagini reali per valutare la capacità di riconoscimento degli annotatori. Questi ultimi hanno mostrato una certa variabilità nelle valutazioni, anche se sono stati mediamente coerenti, inoltre è stato notato che le persone tendono a dare voti peggiori con il tempo in quanto migliorano nel riconoscere le immagini generate.

Un miglioramento dell'**Inception score** è stato proposto successivamente da Martin Heusel et al. [14] nel 2018, i quali hanno proposto un sistema non più basato sull'entropia della distribuzione delle classi predette, ma bensì sulle *features* estratte da un layer interno del modello di classificazione, le quali vengono confrontate tra le immagini generate e le immagini del test set. Tale metodo è stato chiamato **Fréchet Inception Distance** (FID).

### 4.5.1 Fréchet Inception Distance (FID)

La Fréchet Inception Distance è un metodo di valutazione dei risultati ottenuti da un modello di generazione di immagini, il quale si basa sulle *features* estratte da un modello convoluzionale in uno specifico layer e non più dall'output del modello, il quale tipicamente è addestrato sul dataset **ImageNet** per la classificazione. Come accennato la maggiore innovazione portata da questa tecnica sta nel fatto che rispetto all'**Inception score** non si basa più sull'entropia della distribuzione delle classi predette ma su una effettiva comparazione tra la distribuzione reale e quella generata che chiameremo rispettivamente $p_r$ e $p_g$.

Per rendere il calcolo della distanza non troppo complesso si fa l'assunzione che le distribuzioni $p_r$ e $p_g$ siano gaussiane multivariate, dunque la distanza di Fréchet $d(.,.)$ tra le due distribuzioni può essere calcolata come la distanza tra le medie e le matrici di covarianza delle due distribuzioni, ed è definita come segue:

$$d(p_r, p_g) = \|\mu_r - \mu_g\|_2^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}) \tag{4.8}$$

I due termini principali di tale equazione sono uno la distanza tra le medie delle features $\|\mu_r - \mu_g\|_2^2$, mentre l'altro rappresenta la distanza tra le matrici di covarianza delle due distribuzioni $Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$.
I vari termini dell'equazione sono definiti di seguito:

- $\mu_r$: media delle *features* estratte dal modello InceptionV3 sul test set.
- $\mu_g$: media delle *features* estratte dal modello InceptionV3 sulle immagini generate dal modello.
- $\Sigma_r$: matrice di covarianza delle *features* estratte dal modello InceptionV3 sul test set.
- $\Sigma_g$: matrice di covarianza delle *features* estratte dal modello InceptionV3 sulle immagini generate dal modello.
- $Tr$: traccia della matrice.
- $\|\cdot\|_2$: norma euclidea.

Anche per questo metodo è stato utilizzato il medesimo numero di immagini per generare le statistiche necessarie al calcolo della distanza, ovvero 50000 immagini, questa volta però sono state necessarie 50000 immagini generate dal modello e altrettante immagini del test set.

Logicamente possiamo asserire che come una valutazione data da una persona potrebbe essere estremamente soggettiva, così anche una valutazione data da un modello convoluzionale sarà dipendente dall'addestramento del modello stesso. Infatti si è osservato che modelli addestrati su task di classificazione ad esempio tendono a focalizzarsi su *features* di basso livello, dunque textures e colori, mentre modelli addestrati su task di segmentazione tendono a focalizzarsi su *features* di alto livello, dunque forme e strutture più complesse, tale propensione dunque si potrebbe osservare anche nel calcolo della distanza tra le distribuzioni di due insiemi di immagini, utilizzando tali modelli.

### 4.5.2 I risultati

Come risultato finale del progetto si è utilizzata la metrica appena presentata (la FID) per valutare i risultati ottenuti dal modello. L'effettiva procedura utilizzata è una rielaborazione dell'algoritmo originale presentato da Martin Heusel et al. [14] nel 2018, la quale è stata implementata con Pytorch e numpy.

Per la valutazione sono stati utilizzati i dataset di train e di test in formato tile, ovvero immagini di dimensione 256x256, e un ulteriore dataset generato dal modello addestrato. La prima operazione è stata quella di confrontare la FID tra il set di train e il set di test (tiled), per verificare che non ci fossero differenze significative tra le due distribuzioni, ed avere un valore di riferimento. Tale valutazione ha portato ad un punteggio **FID: 1.02** con 40k immagini per il training set e 40k immagini per il test set.

Il risultato ottenuto dal modello è stato abbastanza alto, infatti si è ottenuto un punteggio **FID: 62.79** confrontando 40k immagini del test set con 40k immagini provenienti dal modello, considerando però che i difetti utilizzati per addestrare il discriminatore dei difetti erano in totale 10173 e che 2 di 4 classi avevano meno di 1000 esempi. Tale risultato sottolinea diverse problematiche, le quali hanno portato a un risultato non ottimale, come ad esempio la presenza di *artifacts* che se pur ridotta è ancora presente in qualche caso, e in maniera più frequente si manifestano le variazioni di colore. Tali problemi potrebbero essere legati alla scarsità di dati, ma probabilmente con opportune modifiche alla training pipeline e al modello sarà sicuramente possibile ottenere risultati migliori.

Facciamo a questo punto un confronto con i valori di FID ottenuti da altri modelli SOTA per vari task di generazione basati su architettura GAN:

- **stylegan 2:** su FFHQ per il task di generazione di volti umani ha ottenuto un punteggio di **FID: 2.84**. Va considerato però che il training set consisteva di ben 70k immagini.
- **LaMa:** per il task di inpainting non condizionato ha ottenuto rispettivamente una **FID: 2.21** sul dataset Places con maschere di inpainting estese e una **FID: 7.26** sul dataset CelebA-HQ con maschere di inpainting estese.

# Capitolo 5

# Conclusioni e sviluppi futuri

In conclusione il modello ottenuto alla fine dell'addestramento non ha raggiunto la qualità dei difetti generati sperata, uno dei fattori che probabilmente ha influito negativamente, come già detto, è la dimensione del dataset, il quale è risultato essere troppo piccolo per poter essere utilizzato in una pipeline di questo tipo. Tale carenza si è notata particolarmente per le classi che contavano un minor numero di esempi mentre per le altre gli oggetti generati hanno una buona struttura, anche se non sono utilizzabili per l'addestramento di un modello di segmentazione.

Una considerazione importante derivata dagli esperimenti fatti con la struttura del modello LaMa-fourier è che questa malgrado nel task di inpainting puro abbia dimostrato eccezionali qualità, non sembra ottenere le stesse performance nel caso dell'inpainting controllato. Nel caso dell'inpainting puro infatti, in cui si deve cercare di riprodurre l'immagine originale nell'area mancante, LaMa-fourier riesce efficacemente ad estrarre le caratteristiche delle aree non rimosse già dai primi layer per utilizzarle nella ricostruzione, tale caratteristica però non sembra essere sufficiente ad ottenere un una buona generalizzazione nel task dell'inpainting controllato con un dataset di pochi esempi.

## 5.1 Sviluppi futuri

Un'idea per migliorare le prestazioni del framework, sul quale il progetto verterà se verrà continuato è sicuramente quella di partire dalla struttura del modello **Stylegan 2**, presentato da Tero Karras et al. [18], il quale ha dimostrato eccezionali capacità di generalizzazione e di generazione di immagini di alta qualità.
Una potenziale struttura potrebbe essere sempre basata su layer convoluzionali, ma con le *style injection* attraverso i layer AdaIn proposti in **Stylegan 2**, e utilizzando la *Fast Fourier Convolution* utilizzata in **LaMa-fourier** sempre utilizzando una struttura a encoder-decoder U-net like, tali miglioramenti dovrebbero apportare una maggiore capacità di generalizzazione anche con dataset relativamente piccoli.

# Elenco delle figure

1.1 Un esempio preso dal **Severstal steel defect dataset** di difetti segmentati. credits: Neven Robby and Goedemé Toon, 2021, A Multi-Branch U-Net for Steel Surface Defect Type and Severity Segmentation. https://www.mdpi.com/2075-4701/11/6/870 . . . 2

1.2 Esempio di architettura a solo decoder. . . . 4

1.3 Esempio di architettura con encoder e decoder. . . . 5

1.4 Un diagramma di ven che illustra le relazioni tra i diversi sottogruppi dell'intelligenza artificiale, vediamo infatti come il deep learning sia un sottogruppo del representation learning, che a sua volta è un sottogruppo del machine learning. credits: Yoshua Bengio, Ian J. Goodfellow, Aaron Courville 2015, From the book "Deep Learning" . . . 6

1.5 Schema di un neurone biologico. . . . 7

1.6 Esempio di rete neurale feedforward. In tale rete è possibile vedere i neuroni rappresentati dai nodi del grafo, e le interconnessioni tra di essi che definiscono il peso di ogni relazione, con in rosso un peso positivo e in blu un peso negativo, l'intensità del colore indica la forza della relazione. . . . 8

1.7 Esempio di Neurone artificiale. . . . 8

1.9 Esempio di retropropagazione su di una rete semplificata a 2 strati. . . . 12

1.10 L'immagine raffigura schematicamente il funzionamento di Neocognitron. credits: Kunihiko Fukushima 1980 [7]. . . . 14

1.11 L'immagine tratta dallo stesso articolo illustra schematicamente l'architettura del modello TDNN. credits: Alex Waibel et al. 1989 [29]. . . . 14

1.12 L'immagine tratta dal medesimo articolo illustra l'architettura proposta. credits: Yann LeCun et al. 1989 [19]. . . . 15

1.13 L'immagine illustra l'operazione di convoluzione. credits: Wikipedia. https://en.wikipedia.org /wiki/Convolution . . . 16

1.14 Operazione di convoluzione 2D discreta, in blu abbiamo una matrice 5x5 che rappresenta un'immagine, mentre in verde il risultato di una convoluzione, con un kernel 3x3. I valori del kernel sono raffigurati in basso a destra delle caselle interessate dalla convoluzione. credits: Dumoulin et al. 2016 [5]. . . . . . . . . . 16

1.15 Immagine di Lena Forsén, prima e dopo la convoluzione con i due kernel, che amplificano i bordi verticali e orizzontali. . . . . . . . . . . . . . . . . . . . . . . . . 18

1.16 Esempio di convoluzione con un kernel 3x3, stride pari a 2, e padding (zero padding) pari a 1. credits: Dumoulin et al. 2016 [5] . . . . . . . . . . . . . . . . 19

1.17 Esempio di max pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0. credits: Dumoulin et al. 2016 [5] . . . . . . . . . . . . . . . . . . . . . . 21

1.18 Esempio di average pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0. credits: Dumoulin et al. 2016 [5] . . . . . . . . . . . . . . . . . . . . . . 21

1.19 Esempio di convoluzione di un'immagine rgb. credits: Irhum Shafkat [27] . . . . 22

1.20 Esempio di convoluzione multichannel. credits: Irhum Shafkat [27] . . . . . . . . 22

2.1 Rappresentazione grafica della training pipeline di un modello GAN. . . . . . . . 24

2.2 Confronto tra MSE e Adversarial training. credits: Goodfellow [10] . . . . . . . . 25

2.4 Visualizzazione schematica della convergenza della distribuzione dei dati generati verso quelli reali, e dell'uscita del discriminatore al variare di $\mathbf{x}$. Si noti che la linea nera tratteggiata rappresenta la distribuzione dei dati reali $p_{data}$, mentre la linea verde continua rappresenta la distribuzione dei dati generati $p_{model}$ e in fine la linea blu tratteggiata rappresenta l'uscita del discriminatore $D(x)$, al variare di $\mathbf{x}$. I 2 assi in fondo sono rispettivamente, l'asse $z$ i valori casuali di input del generatore con distribuzione fissa, e l'asse $x$, i valori di input del discriminatore, appartenenti alla distribuzione $\mathbf{p_{data}}$ o $\mathbf{p_g}$. credits: Goodfellow et al. [11] . . . . 27

2.6 Esempio di corretto apprendimento (prima riga) e di mode collapse (seconda riga). credits: Luke Metz et al. [22] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.7 Alcuni risultati del modello addestrato in questa ricerca, relativi a un dataset di facce umane in bassa risoluzione (b) e il dataset Minst (a). credits: Goodfellow et al. [11] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.8 Architettura del generatore di DCGAN. credits: Alec Radford et al. [24] . . . . . 30

2.9 Esempio di algebra vettoriale nello spazio latente Z. credits: Alec Radford et al. [24] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.10 Esempio di *gradient vanishing*, causato da un discriminatore addestrato fino all'ottimo per un generatore non ottimo. . . . . . . . . . . . . . . . . . . . . . . . 32

2.11 In questa immagine è possibile vedere la derivata della funzione seno, nei punti $\pi/4$, $\pi/2$ e $\pi$, ed è possibile vedere come tale funzione rispetta la *1-Lipschitz continuity*, essendo la sua derivata all'interno dell'intervallo ammesso in ogni punto. 34

2.12 In questa immagine è possibile vedere la derivata della funzione $x^2$, nei punti 0.2, 0.4 e 0.8, in questo caso la funzione non rispetta la *1-Lipschitz continuity*, in quanto la sua derivata cresce rapidamente oltre il limite consentito dopo x=0.4. . 34

2.13 In questa immagine è possibile vedere come il critico $C(x)$ sia in grado di mappare efficacemente la distanza tra due distribuzioni $P_{data}$ e $P_G$, restituendo dei gradienti utili per il generatore anche quando il critico è addestrato all'ottimo per il dato generatore. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.14 Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Wasserstein distance*. credits: Martin Arjovsky et al. [1] . . . . . . . . . . . . . . 35

2.15 Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Jensen-Shannon divergence*. credits: Martin Arjovsky et al. [1] . . . . . . . . . . 35

2.16 In questa immagine è possibile vedere un esempio di training di Pix2Pix per il task *edges to photo*, In tale configurazione il discriminatore apprende come discernere tra coppie di immagini e schizzi, e si può vedere come diversamente dal caso classico sia il generatore che il discriminatore ricevono in ingresso l'immagine di riferimento (lo schizzo) credits: Phillip Isola et al. [15] . . . . . . . . . . . . . . . 37

2.17 In questa immagine è illustrata intuitivamente la differenza tra un comune modello ad *encoder-decoder* e l'architettura di U-net. credits: Phillip Isola et al. [15] . . . 37

2.18 In questa immagine è possibile vedere una comparazione di un sample generato da 4 diversi modelli per il task *labels to scene*, i quattro esempi illustrano l'utilizzo o meno delle *skip connection* e l'utilizzo di una loss L1 a una loss L1 più *conditional adversarial loss*. credits: Phillip Isola et al. [15] . . . . . . . . . . . . . . . . . . . 38

2.19 In questa immagine sono mostrati alcuni esempi di applicazione di pix2pix. credits: Phillip Isola et al. [15] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

2.20 In questa immagine è illustrata schematicamente la struttura di LaMa. credits: Roman Suvorov et al. [28] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

2.21 In questa immagine sono mostrati schematicamente i passaggi che permettono di calcolare la perceptual loss da un modello convoluzionale. credits: Richard Zhang et al. [31] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

2.22 In questa immagine sono mostrati i risultati ottenuti da Mescheder et al. in [21] utilizzando la R1 regularization. In blu è rappresentata la distribuzione di riferimento, in arancione i campioni generati, mentre da viola a giallo si ha l'intensità del gradiente di $D$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

2.23 In questa immagine sono mostrati alcuni risultati ottenuti da LaMa. In blu sono rappresentate le maschere applicate alle immagini, e dunque corrispondenti alle arre che sono state rimosse prima della propagazione nella rete per la ricostruzione. Le immagini sulla destra rappresentano l'output del modello. credits: Roman Suvorov et al. [28] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.1 Figura che illustra un esempio di input di COIGAN, il quale è un tensore di 7 canali, in questo caso con un'immagine RGB e 4 canali per le maschere che effettuano il condizionamento della generazione dei difetti. . . . . . . . . . . . . . 56

4.2 Figura che illustra in maniera riassuntiva le componenti principali della pipeline di addestramento di COIGAN. Le frecce rappresentano la propagazione dei vari tensori, mentre le linee rosse indicano la propagazione dei gradienti delle loss. . . 56

4.3 Funzione che effettua la decodifica di una maschera codificata in RLE. . . . . . 58

4.4 Esempio di file contenente gli indici degli esempi. . . . . . . . . . . . . . . . . . . 59

4.5 Esempio di file contenente le annotazioni nel formato *line json*. . . . . . . . . . . 59

4.7 Figura che illustra il processo di creazione dei dataset degli oggetti. In questo caso viene mostrato come vengono estratti i difetti da una singola immagine contenente 3 difetti, i quali vengono smistati nei rispettivi dataset. . . . . . . . . . . . . . . . . 61

4.8 Esempio di file contenente le annotazioni degli object dataset nel formato *line json*. 61

4.9 Figura che illustra concettualmente il processo di tiling delle immagini. . . . . . . 62

4.10 In Figura è mostrato il diagramma UML semplificato del dataloader e delle classi principali che lo compongono. Questo infatti suddivide diverse operazioni di caricamento dei dati in classi separate, le quali fanno convergere i risultati all'oggetto **CoiganSeverstalSteelDefectsDataset** che restituisce un output compatibile con l'oggetto **torch.utils.data.DataLoader**. . . . . . . . . . . . . . 64

4.11 Figura che illustra concettualmente il processo di caricamento degli oggetti da parte del **ObjectDataloader**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4.12 Figura che illustra un esempio di applicazione del **GassianNoiseGenerator**. . . 67

4.13 Figura che illustra un esempio di generazione di rumore all'interno del **MultiscaleNoiseGenerator**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.14 Figura che illustra un esempio di applicazione del rumore generato dal **MultiscaleNoiseGenerator**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.15 Nella figura è mostrato il contenuto di un sample che si può ottenere dall'oggetto **CoiganSeverstalSteelDefectsDataset**, a sinistra i cubi rappresentano i tensori, con i nomi effettivi che sono presenti nel dict, mentre a destra vi sono i canali dei tensori, le immagini raggruppano 3 canali mentre le maschere occupano un solo canale. Si noti che l'ordine da sinistra verso destra è coerente con l'ordine dei canali nei tensori. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

4.16 Figura che illustra graficamente il processo di calcolo della loss L1 smooth masked. 73

4.17 Figura che illustra concettualmente la struttura interna del discriminatore basato sulla trasformata *wavelet*. credits: Rinon Gal et. al [8] . . . . . . . . . . . . . . . . 76

4.18 Figura che illustra concettualmente il processo di estrazione dei difetti dall'immagine generata, per l'addestramento del discriminatore. . . . . . . . . . . . . . . . 77

4.19 Figura che illustra un batch di immagini generate, da una precedente versione della training pipeline, non provvista di **ref discriminator**, e con le loss **L1** e **Perceptual loss** che non utilizzano lo smooth della weight mask. . . . . . . . . 78

4.20 Figura che illustra un batch di immagini generate, da una versione della training pipeline provvista di **ref discriminator**, e con le loss **L1** e **Perceptual loss** che utilizzano lo smooth della weight mask. . . . . . . . . . . . . . . . . . . . . . . . . 79

# Bibliografia

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[2] World Steel Association. World steel in figures 2021, 2021.

[3] Lu Chi, Borui Jiang, and Yadong Mu. Fast fourier convolution, 2020.

[4] George Cybenko. Approximation by superpositions of a sigmoidal function, 1989.

[5] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[6] Maayan Frid-Adar, Idit Diamant, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification, 2018.

[7] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, 1980.

[8] Rinon Gal, Dana Cohen, Amit Bermano, and Daniel Cohen-Or. Swagan: A style-based wavelet-driven generative model, 2021.

[9] Golnaz Ghiasi, Yin Cui, Aravind Srinivas, Rui Qian, Tsung-Yi Lin, Ekin D. Cubuk, Quoc V. Le, and Barret Zoph. Simple copy-paste is a strong data augmentation method for instance segmentation, 2021.

[10] Ian J. Goodfellow. Introduction to generative adversarial networks, 2016.

[11] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[12] Anatoli Gorchet. Deep learning has reinvented quality control in manufacturing—but it hasn t gone far enough, 2020.

[13] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks, 2015.

[14] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

[15] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.

[16] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016.

[17] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019.

[18] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition, 1989.

[20] Yang Lei, Yabo Fu, Tonghe Wang, Richard L. J. Qiu, Walter J. Curran, Tian Liu, and Xiaofeng Yang. Deep learning in multi-organ segmentation, 2020.

[21] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge?, 2018.

[22] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks, 2017.

[23] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

[24] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[26] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[27] Irhum Shafkat. Intuitively understanding convolutions for deep learning, 2018.

[28] Roman Suvorov, Elizaveta Logacheva, Anton Mashikhin, Anastasia Remizova, Arsenii Ashukha, Aleksei Silvestrov, Naejin Kong, Harshith Goka, Kiwoong Park, and Victor Lempitsky. Resolution-robust large mask inpainting with fourier convolutions, 2021.

[29] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. Phoneme recognition using time-delay neural networks, 1989.

[30] Vivian Wen Hui Wong, Max Ferguson, Kincho H. Law, Yung-Tsun Tina Lee, and Paul Witherell. Automatic volumetric segmentation of additive manufacturing defects with 3d u-net, 2021.

[31] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018.

[32] Lili Zhu, Petros Spachos, Erica Pensini, and Konstantinos Plataniotis. Deep learning and machine vision for food processing: A survey, 2021.