



UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

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

**Sviluppo di un modello GAN per la generazione di
immagini e relativa segmentazione**

Develop of a GAN model for image generation with related segmentations

Relatore:

Prof. Adriano Mancini

Laureando:

Massimiliano Biancucci

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	ii
Indice	iii
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
2.5	LAMA	37
3	Materiali e metodi	38
3.1	Il Dataset: Severstal steel defect detection	38
3.1.1	Distribuzione del dataset	39
3.2	Librerie e Framework	43
3.2.1	Numpy	43
3.2.2	OpenCV	43
3.2.3	Pytorch	43
3.2.4	Distributed data parallel	43
3.3	Google cloud compute instance	44
3.4	Repository del progetto	44
4	Sviluppo del progetto	45
4.1	Definizione della pipeline di addestramento	45
4.2	Preparazione dei dati per la pipeline	45
4.3	Il dataloader	45
4.4	Il trainer	45
4.5	Lo script di evaluation	45
4.6	Tool per l'inferenza interattiva	45

5 Risultati	46
Elenco delle figure	47
Bibliografia	50

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 [19], questo tipo di task è utilizzato ad esempio nel campo medico per la segmentazione di organi [16], tumori [11], o in campo industriale per la segmentazione di difetti [26], 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 [10, 28], 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.

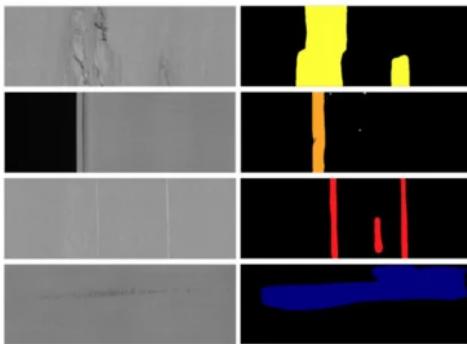


Figura 1.1: Un esempio preso dal **Severstal steel defect dataset** di difetti segmentati.
credits: Neven Robby and Goedemé Toon, 2021, A Multi-Branch U-Net for Steel Surface Defect Type and Severity Segmentation. <https://www.mdpi.com/2075-4701/11/6/870>

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 richiedere 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* [4], 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 [9] , 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*" [7], 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 efficace 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 *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 tenore 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).

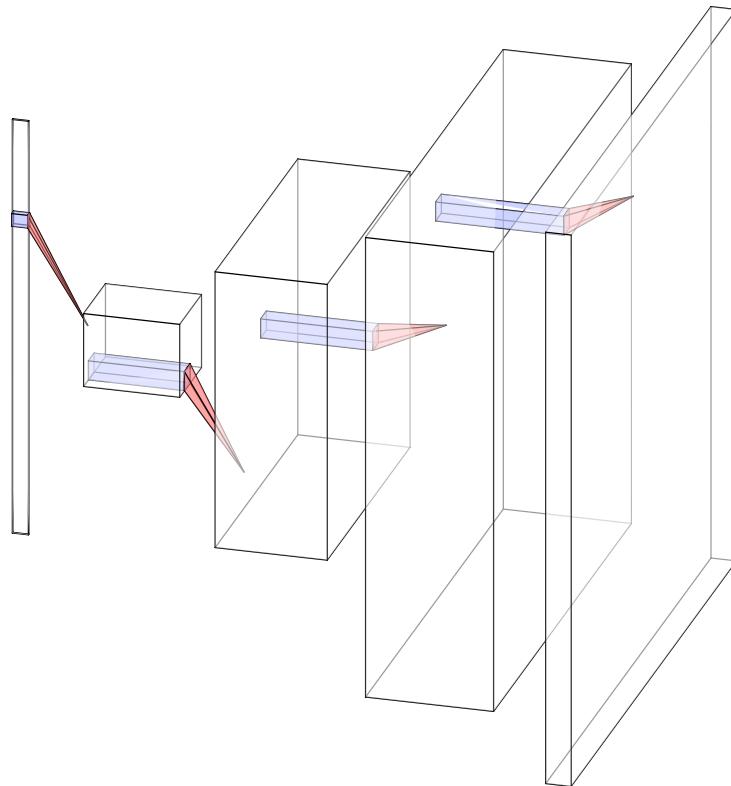


Figura 1.2: Esempio di architettura a solo decoder.

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.

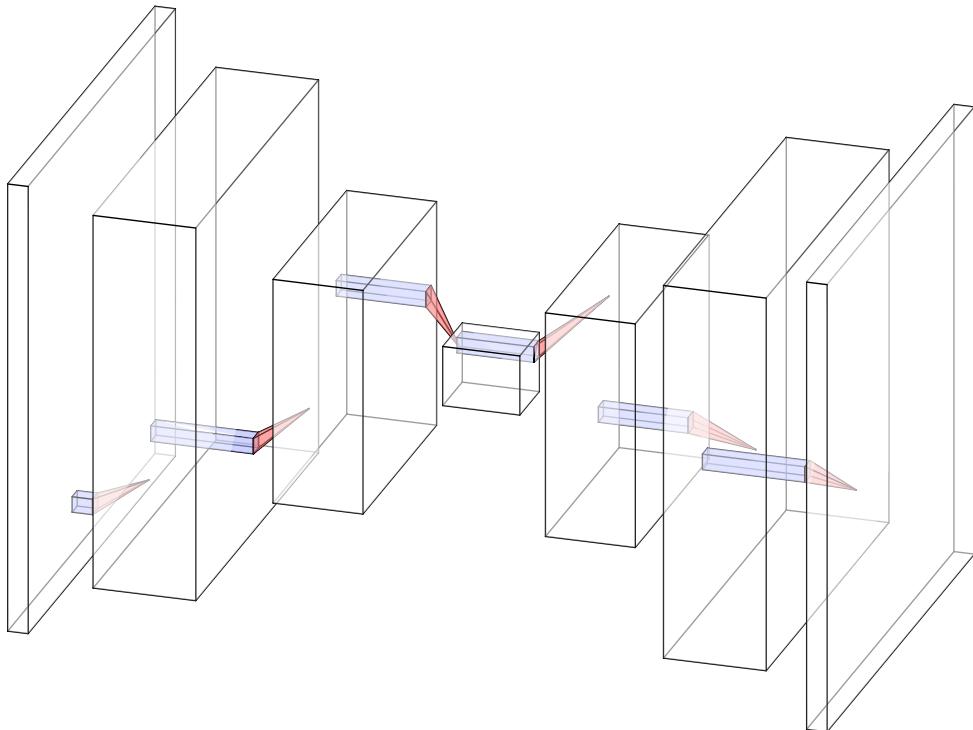


Figura 1.3: Esempio di architettura con encoder e decoder.

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.

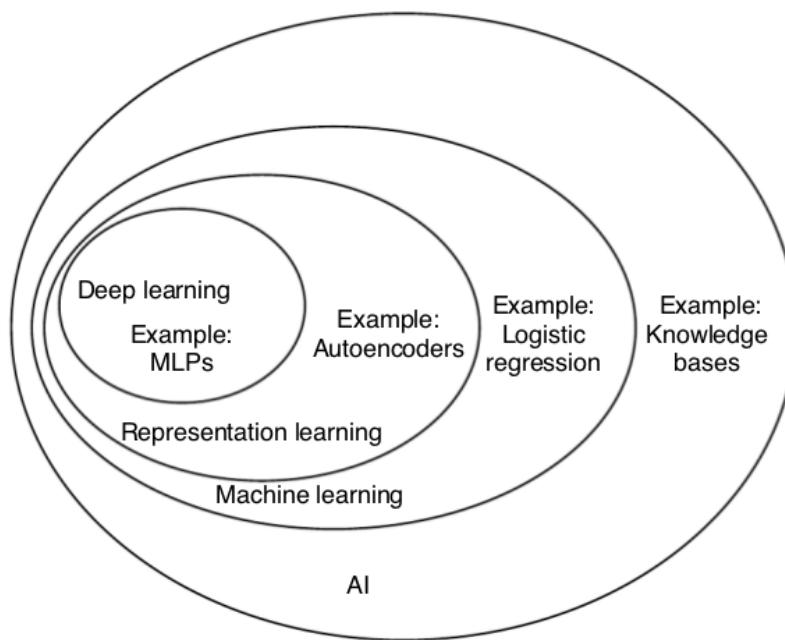


Figura 1.4: Un diagramma di ven che illustra le relazioni tra i diversi sottogruppi dell'intelligenza artificiale, vediamo infatti come il deep learning sia un sottogruppo del representation learning, che a sua volta è un sottogruppo del machine learning.
credits: Yoshua Bengio, Ian J. Goodfellow, Aaron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

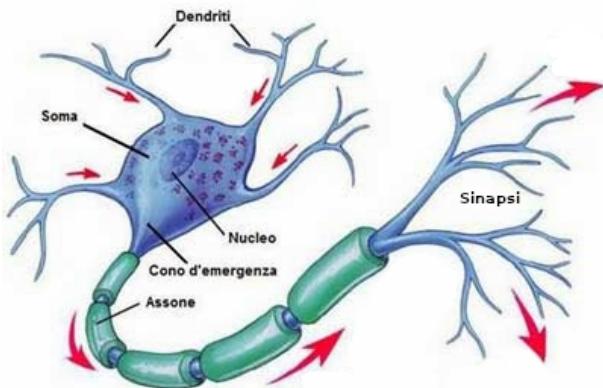


Figura 1.5: Schema di un neurone biologico.

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.

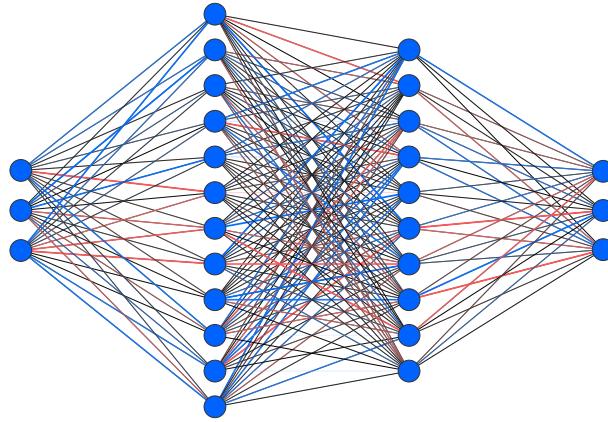


Figura 1.6: Esempio di rete neurale feedforward. In tale rete è possibile vedere i neuroni rappresentati dai nodi del grafo, e le interconnessioni tra di essi che definiscono il peso di ogni relazione, con in rosso un peso positivo e in blu un peso negativo, l'intensità del colore indica la forza della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

1.3.1 Il neurone artificiale

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:

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

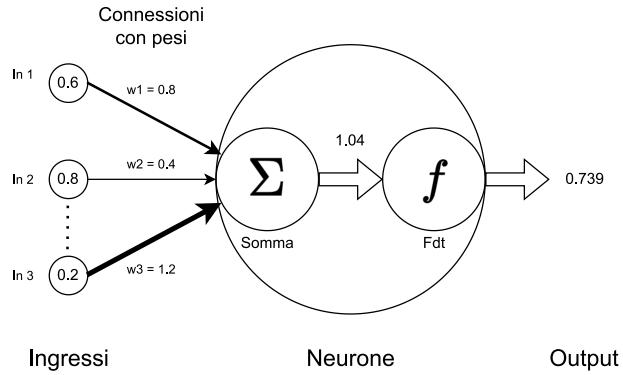
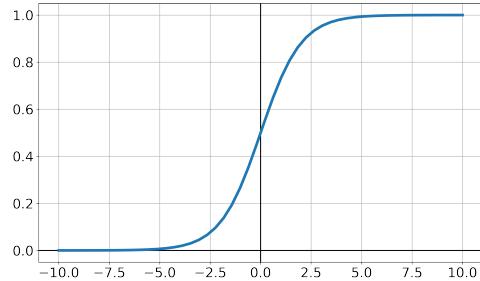


Figura 1.7: Esempio di Neurone artificiale.

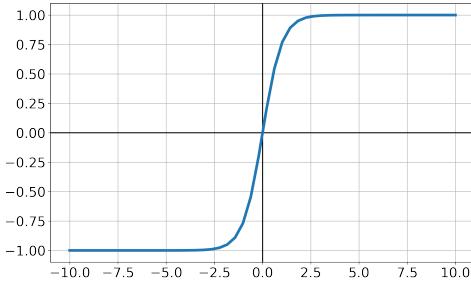
Considerando y l'output del neurone, \vec{w} il vettore dei pesi delle connessioni in ingresso, \vec{x} il vettore degli input, b il bias ovvero un valore aggiunto alla somma pesata degli input dipendente dal neurone e f la funzione di trasferimento. La funzione di attivazione o funzione di trasferimento 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:



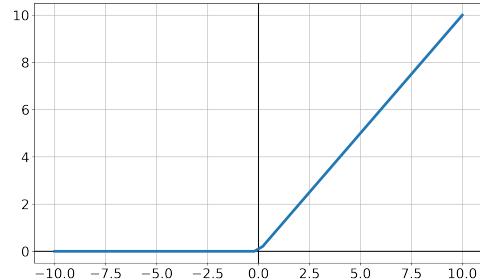
(a) Sigmoids

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



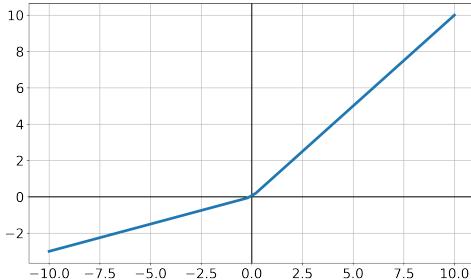
(b) Tangente iperbolica

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



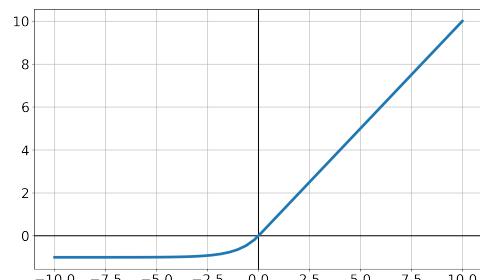
(e) Rectified linear unit (ReLU)

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



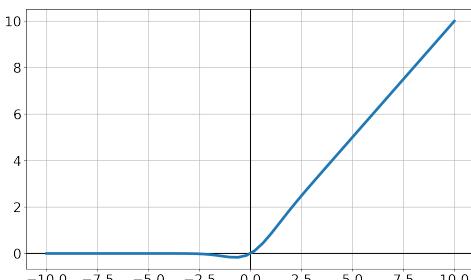
(f) Leaky RELU

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



(i) Exponential linear unit (ELU)

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



(j) Gaussian error linear unit (GELU)

$$f(x) = x * \Phi(x)$$

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, Geoffrey 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 $\mathbf{g}(\mathbf{x})$ come combinazione di composizione di funzioni e moltiplicazioni di matrici.

$$\tilde{\mathbf{y}} = \mathbf{g}(\tilde{\mathbf{x}}) = \mathbf{f}_L(\mathbf{W}^L \cdot \mathbf{f}^{L-1}(\mathbf{W}^{L-1} \cdot \mathbf{f}^{L-2}(\dots \mathbf{W}^3 \cdot \mathbf{f}^2(\mathbf{W}^2 \cdot \mathbf{f}^1(\mathbf{W}^1 \cdot \tilde{\mathbf{x}})) \dots))) \quad (1.2)$$

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 $\tilde{\mathbf{x}}$ e $\tilde{\mathbf{y}}$ sono rispettivamente il vettore di input e il vettore di output della rete neurale, se consideriamo $\hat{\mathbf{y}}$ 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.

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

A questo punto data una determinata coppia di input e output $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$, 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.

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial w_{ij}^l} \quad (1.4)$$

Ottieniamo così il gradiente della funzione di costo per il peso w_{ij}^l 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 $\tilde{\mathbf{z}}^l$ e l'output dei neuroni dopo la funzione di attivazione $\tilde{\mathbf{a}}^l$ per ogni layer.

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

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

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 $\nabla_{\mathbf{a}^L} \mathbf{E}$, invertendo l'ordine dei prodotti e trasponendo le matrici:

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

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

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

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.

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

Quindi:

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

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

...

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

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

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:

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

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 $\nabla_{\mathbf{w}^1} \mathbf{E}$ nel seguente modo:

$$\nabla_{\mathbf{w}^1} \mathbf{E}_{k,k} = \frac{\partial \mathbf{E}}{\partial w_{k,k}^1} \quad (1.10)$$

Tale gradiente a questo punto può essere utilizzato per aggiornare il peso $w_{k,k}^l$, applicando un semplice coefficiente di apprendimento η :

$$\mathbf{W}_{k,k}^1 = \mathbf{W}_{k,k}^1 + \Delta \mathbf{W}_{k,k}^1 \quad (1.11)$$

$$\Delta \mathbf{W}_{k,k}^1 = -\eta \cdot \nabla_{\mathbf{w}^1} \mathbf{E}_{k,k} \quad (1.12)$$

è 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* [2], 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'iper cubo 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 σ qualsiasi funzione sigmoide arbitraria. Le sommatorie finite della forma*

$$\mathbf{G}(\mathbf{x}) = \sum_{i=1}^N \alpha_i \cdot \sigma(\mathbf{y}_i^T \mathbf{x} + \Theta_i) \quad (1.13)$$

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:

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

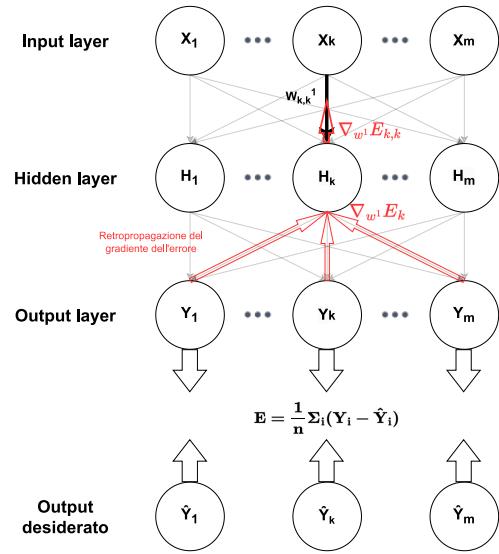


Figura 1.9: Esempio di retropropagazione su di una rete semplificata a 2 strati.

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* [5], 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.

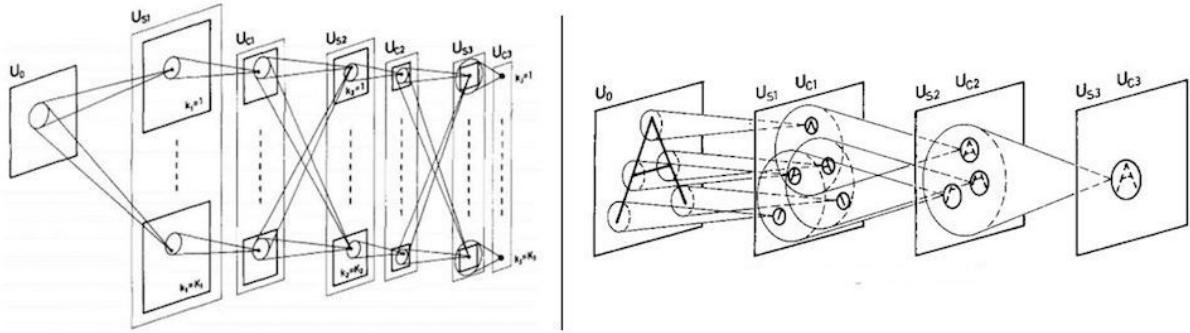


Figura 1.10: L'immagine raffigura schematicamente il funzionamento di Neocognitron.
credits: Kunihiko Fukushima 1980 [5].

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" [25] 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*.

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

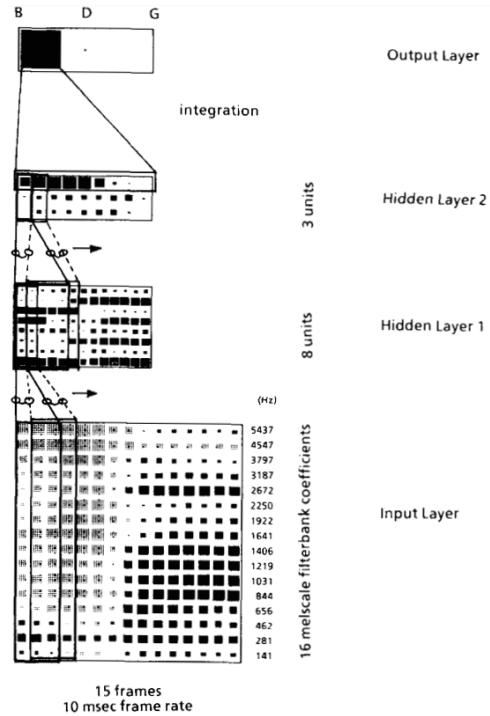


Figura 1.11: L'immagine tratta dallo stesso articolo illustra schematicamente l'architettura del modello TDNN.
credits: Alex Waibel et al. 1989 [25].

delle CNN è stata quella di Yann LeCun et al. "Backpropagation applied to handwritten zip code recognition" [15] 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.

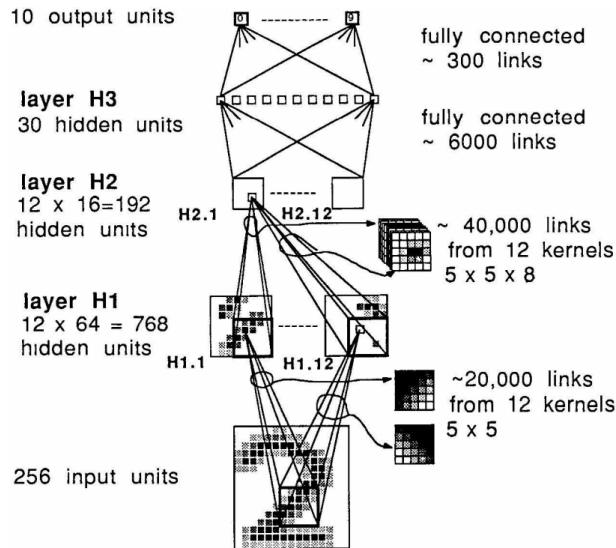


Figura 1.12: L'immagine tratta dal medesimo articolo illustra l'architettura proposta.
credits: Yann LeCun et al. 1989 [15].

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:

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

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:

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

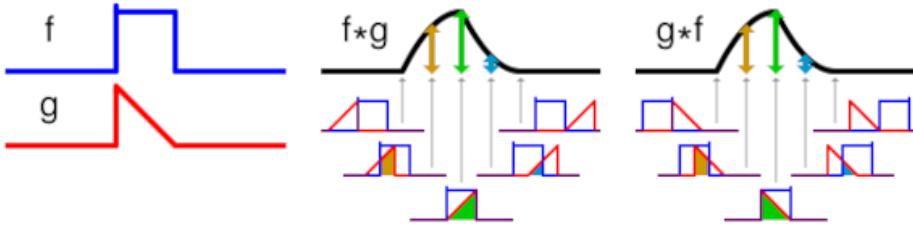


Figura 1.13: L'immagine illustra l'operazione di convoluzione.
credits: Wikipedia. <https://en.wikipedia.org/wiki/Convolution>

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.

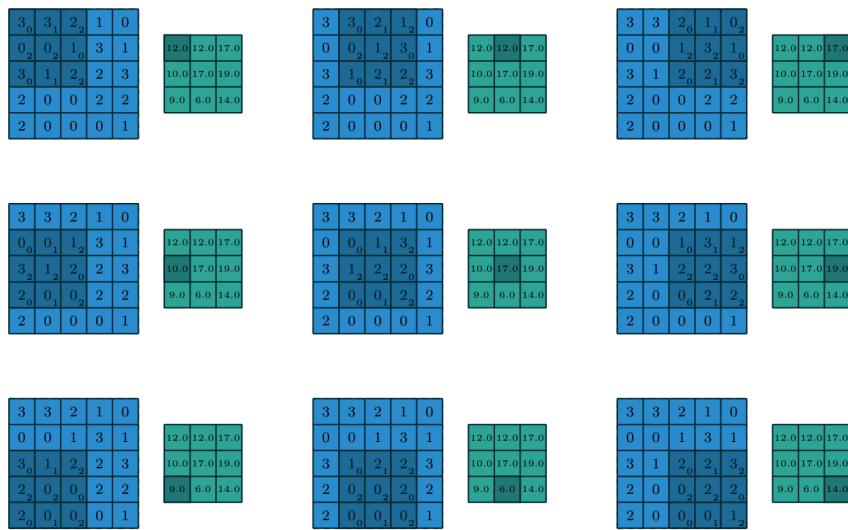


Figura 1.14: Operazione di convoluzione 2D discreta, in blu abbiamo una matrice 5x5 che rappresenta un'immagine, mentre in verde il risultato di una convoluzione, con un kernel 3x3. I valori del kernel sono raffigurati in basso a destra delle caselle interessate dalla convoluzione.
credits: Dumoulin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

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:

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

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

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

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

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.

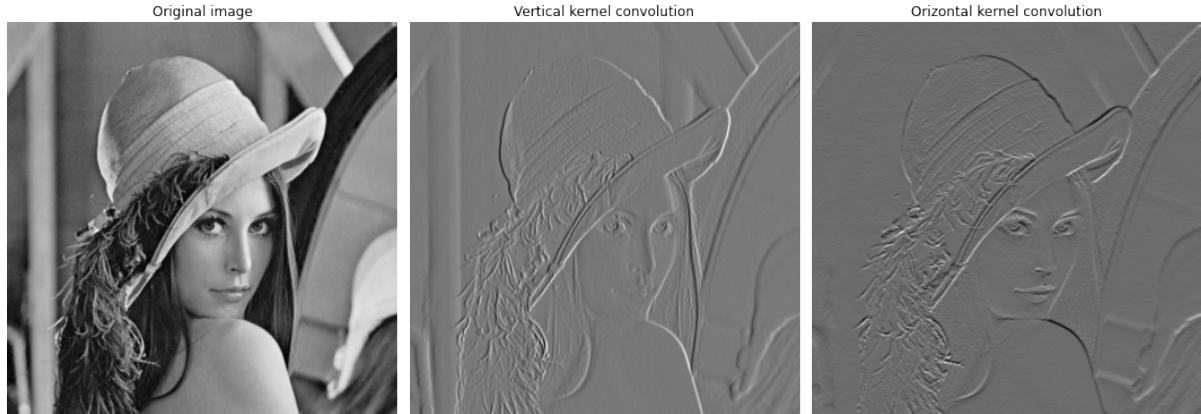


Figura 1.15: Immagine di Lena Forsén, prima e dopo la convoluzione con i due kernel, che amplificano i bordi verticali e orizzontali.

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 sono 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.

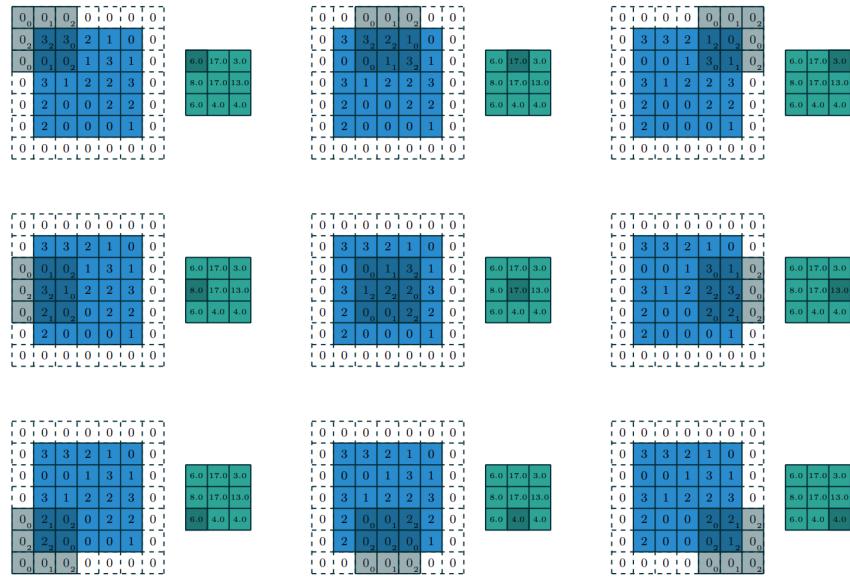


Figura 1.16: Esempio di convoluzione con un kernel 3x3, stride pari a 2, e padding (zero padding) pari a 1.

credits: Dumoulin et al. 2016 [3]

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:

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

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

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:

$$H_{out} = \frac{H_{in} - H_k}{S} + 1 \quad (1.22)$$

$$W_{out} = \frac{W_{in} - W_k}{S} + 1 \quad (1.23)$$

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.

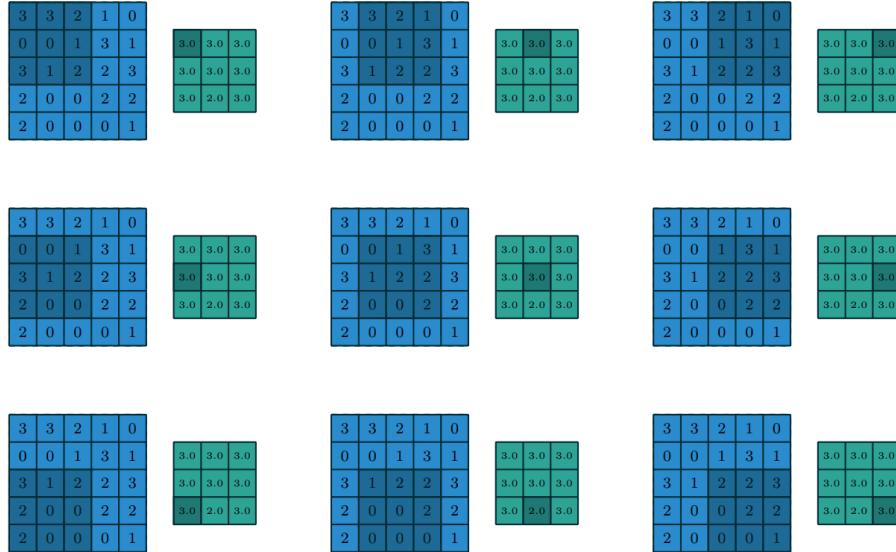


Figura 1.17: Esempio di max pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0.

credits: Dumoulin et al. 2016 [3]

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.

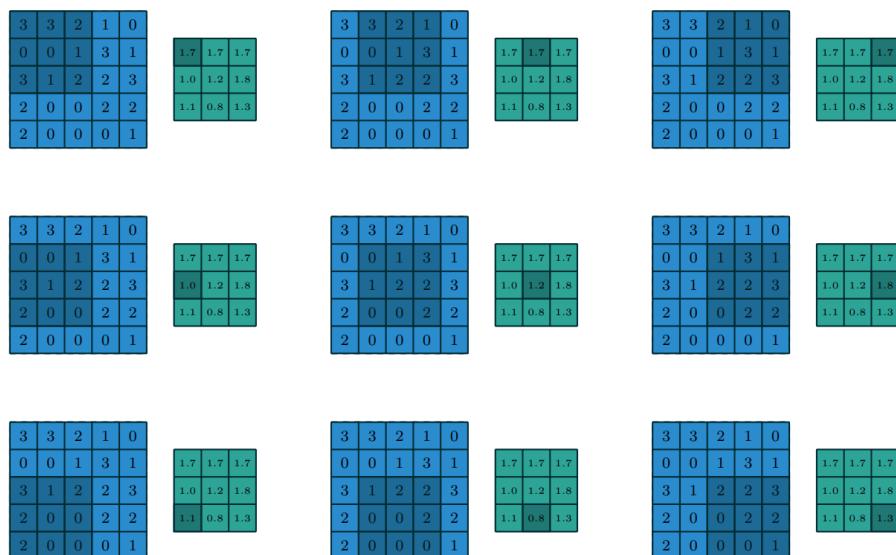


Figura 1.18: Esempio di average pooling con una finestra di dimensione 3x3, stride pari a 1 e padding 0.

credits: Dumoulin et al. 2016 [3]

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.

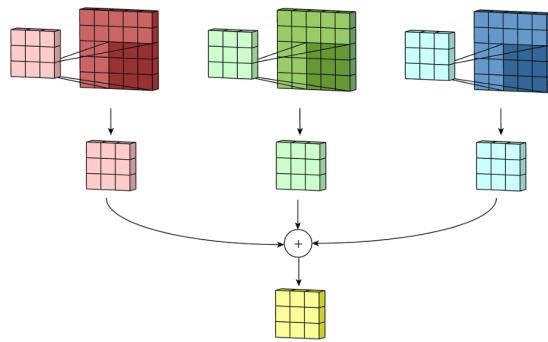


Figura 1.19: Esempio di convoluzione di un'immagine rgb.
credits: Irhum Shafkat [23]

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 volta 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.

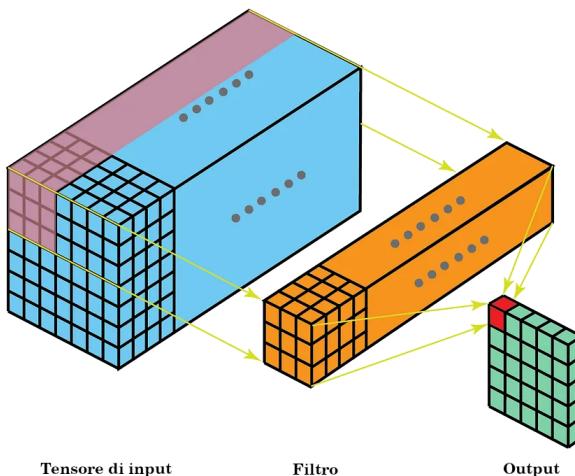


Figura 1.20: Esempio di convoluzione multichannel.
credits: Irhum Shafkat [23]

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. [9] capostipite di questa famiglia di modelli passando per alcune delle principali innovazioni proposte da altri autori, fino ad arrivare al modello LAMA [24], 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. [9] 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 sempre associato 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 **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 caso 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, $\mathbf{x} \in \mathbb{R}^n$.
- \mathbf{z} : input casuale, utilizzato per determinare una mappatura nello spazio dei dati, $\mathbf{z} \in \mathbb{R}^m$.
- \mathbf{y} : uscita del discriminatore, $\mathbf{y} \in \{\mathbf{0}, \mathbf{1}\}$, esprime la probabilità che \mathbf{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.
- θ_G : parametri del generatore.
- θ_D : parametri del discriminatore.
- $\mathbf{G}(\theta_G, \mathbf{z})$: Funzione generatore che mappa lo spazio random z nello spazio dei dati, attraverso i parametri θ_G . definibile come una funzione $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$.
- $\mathbf{D}(\theta_D, \mathbf{x})$: Funzione discriminatore che mappa lo spazio dei dati in un uno spazio $\{\mathbf{0}, \mathbf{1}\}$, attraverso i parametri θ_D . definibile come una funzione $\mathbf{f} : \mathbb{R}^n \rightarrow \{\mathbf{0}, \mathbf{1}\}$. Tale spazio identifica la provenienza di un esempio \mathbf{x} come segue:
 - se $\mathbf{D}(\theta_D, \mathbf{x}) = 1 \rightarrow \mathbf{x} \in \mathbf{p}_{\text{data}}$.
 - se $\mathbf{D}(\theta_D, \mathbf{x}) = 0 \rightarrow \mathbf{x} \in \mathbf{G}(\theta_G, \mathbf{z}), z \sim p_z$.

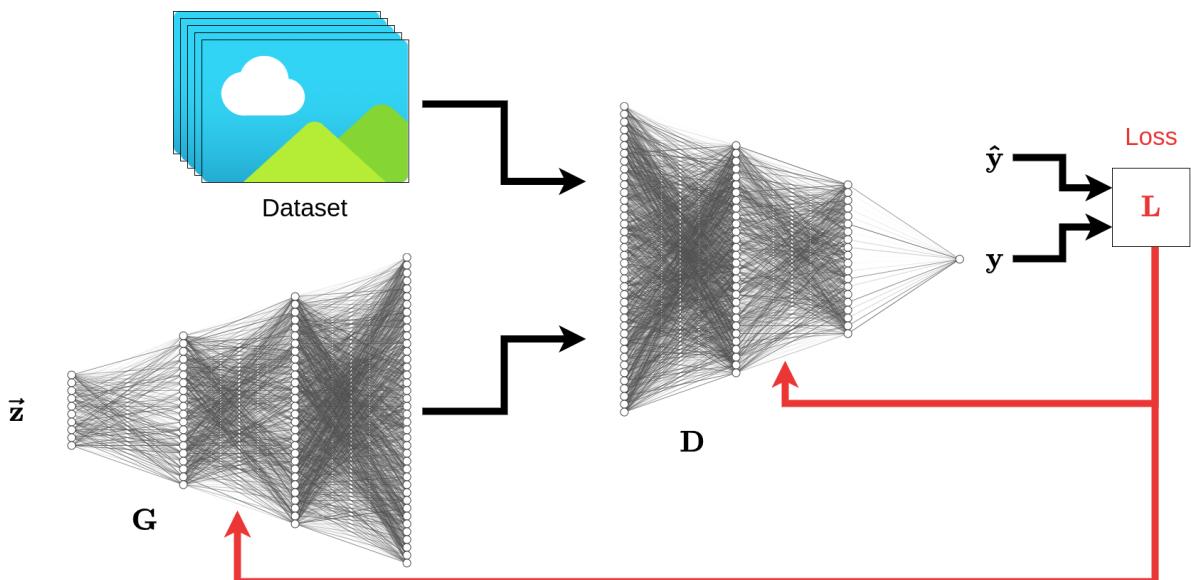


Figura 2.1: Rappresentazione grafica della training pipeline di un modello GAN.

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).

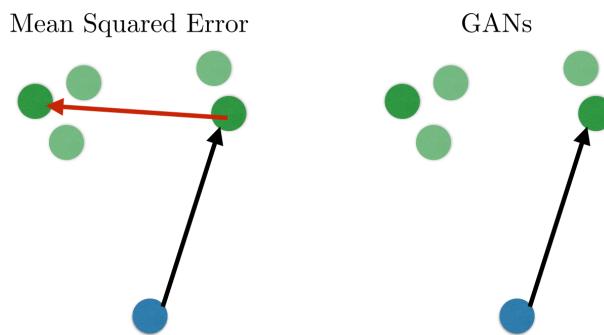


Figura 2.2: Confronto tra MSE e Adversarial training.
credits: Goodfellow [8]

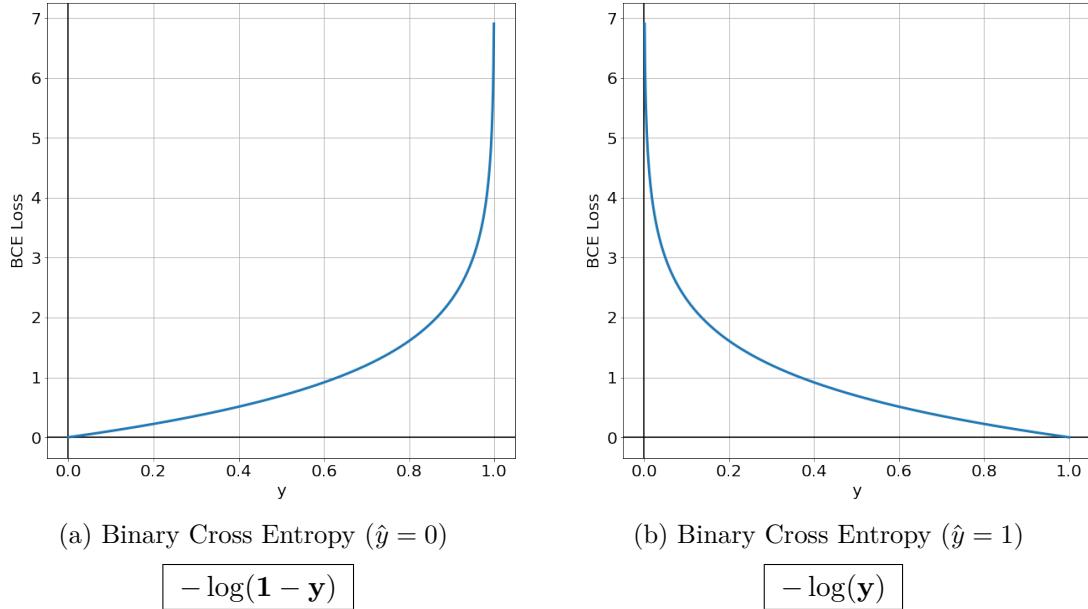
Nella figura possiamo vedere in verde un ipotetico insieme di dati reali e in blu i dati generati dal generatore. Nel caso del MSE vediamo come il gradiente della loss spinge il generatore verso la zona centrale in quanto punta alla media dei dati reali, anche se quella zona non fa parte della distribuzione, 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:

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

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



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

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

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

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

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

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

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

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

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

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 x , ossia della variabile di input del discriminatore, in questo caso unidimensionale ma in generale potrebbe rappresentare immagini, audio video e così via.

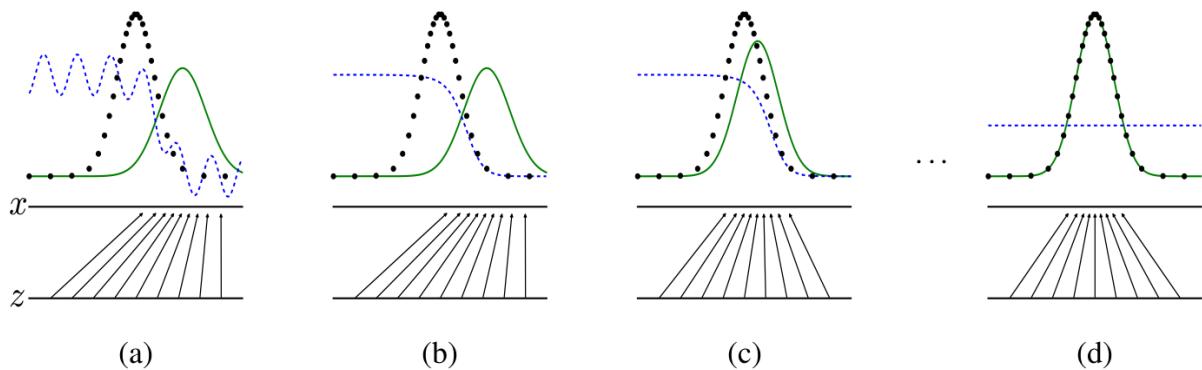


Figura 2.4: Visualizzazione schematica della convergenza della distribuzione dei dati generati verso quelli reali, e dell'uscita del discriminatore al variare di x .

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 x . I 2 assi in fondo sono rispettivamente, l'asse z i valori casuali di input del generatore con distribuzione $\mathbf{P}_{data} \circ \mathbf{P}_g$.

credits: Goodfellow et al. [9]

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:

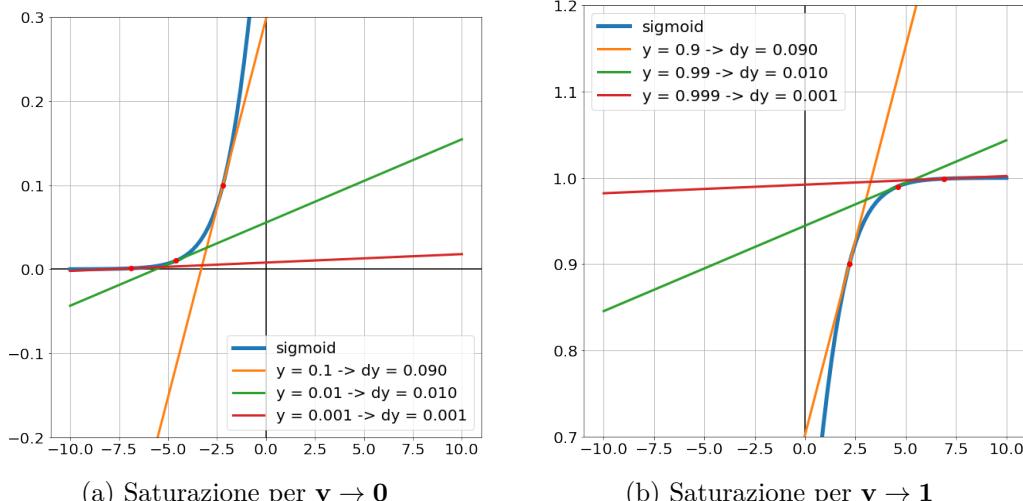
$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \quad (2.7)$$

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

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

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 capitolo precedente anche i gradienti dei layer precedenti tenderanno a zero, e l'addestramento si arresterà. Vediamo di eseguire la funzione di trasferimento sigmoide e il suo gradiente nel momento in cui tende a saturare:



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 p_{data} . Tale condizione consente comunque di raggiungere la condizione di ottimalità sopra descritta con $p_g \subset p_{\text{data}}$, ottenendo comunque $D^*(x) = \frac{1}{2}$. 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.

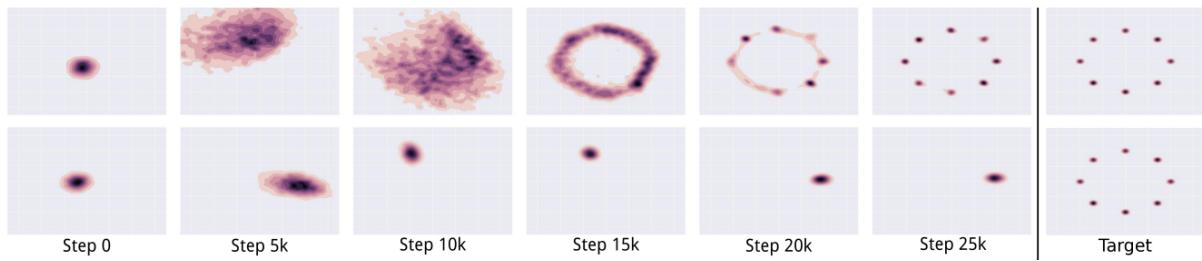


Figura 2.6: Esempio di corretto apprendimento (prima riga) e di mode collapse (seconda riga).
credits: Luke Metz et al. [18]

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.

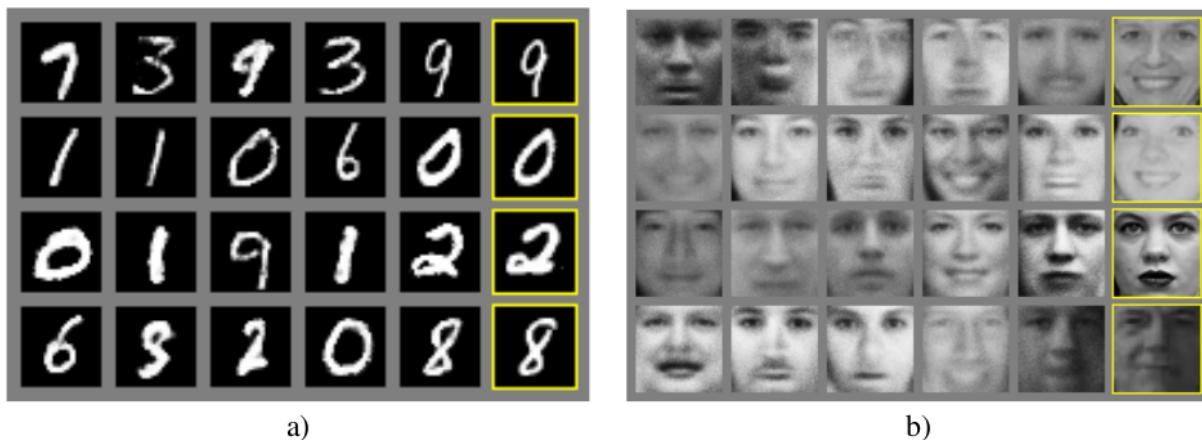


Figura 2.7: Alcuni risultati del modello addestrato in questa ricerca, relativi a un dataset di facce umane in bassa risoluzione (b) e il dataset Minst (a).

credits: Goodfellow et al. [9]

2.2 DCGAN

DCGAN è stata la prima pubblicazione, opera di Alec Radford et. al [20], 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:

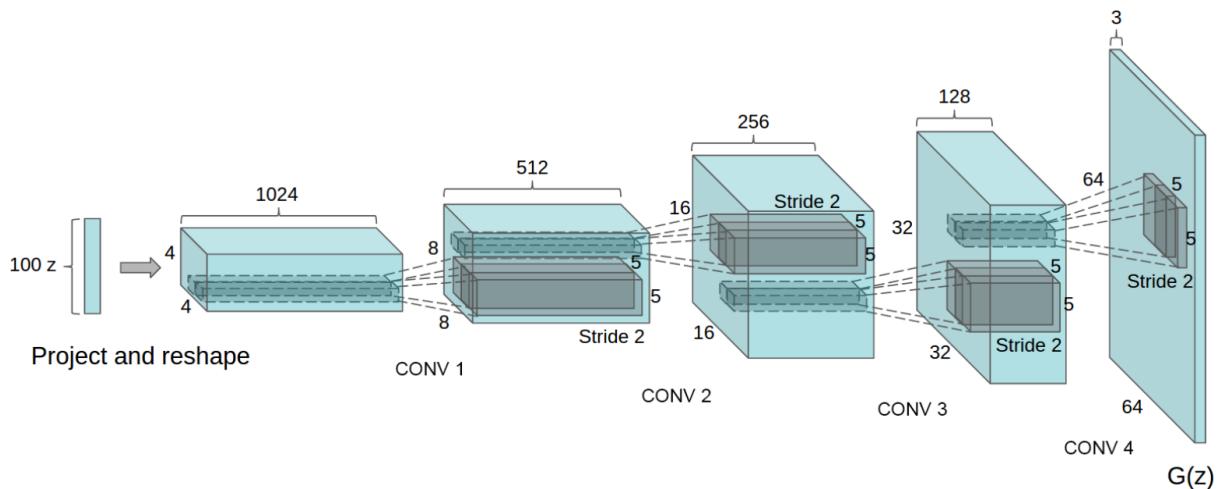


Figura 2.8: Architettura del generatore di DCGAN.
credits: Alec Radford et al. [20]

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.

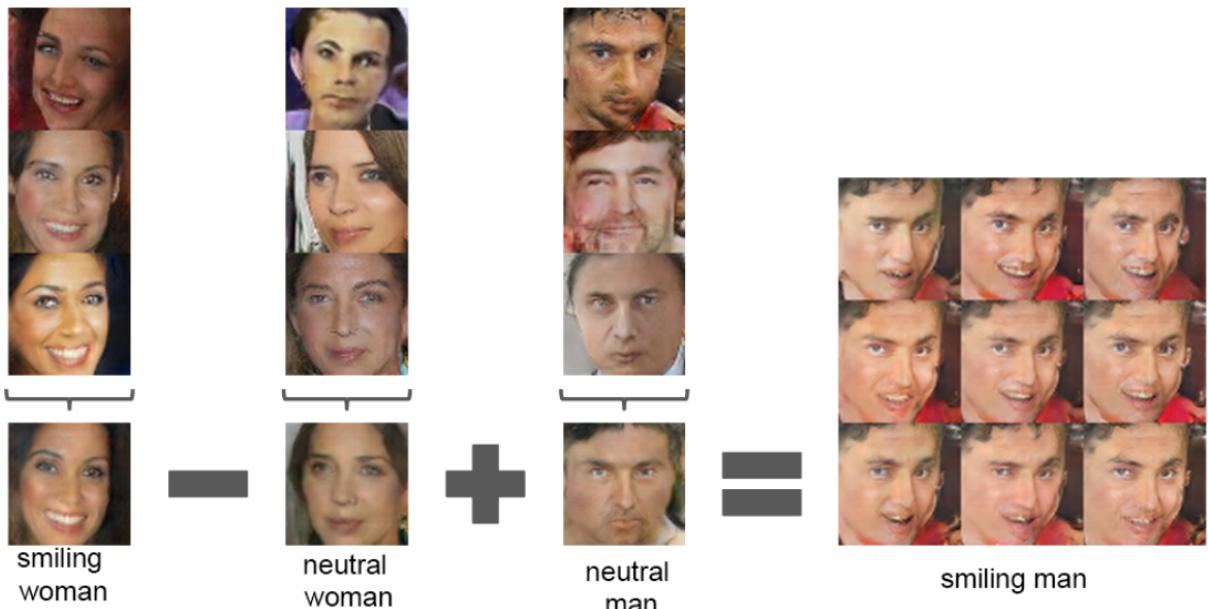


Figura 2.9: Esempio di algebra vettoriale nello spazio latente Z.

credits: Alec Radford et al. [20]

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 efficace 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 .

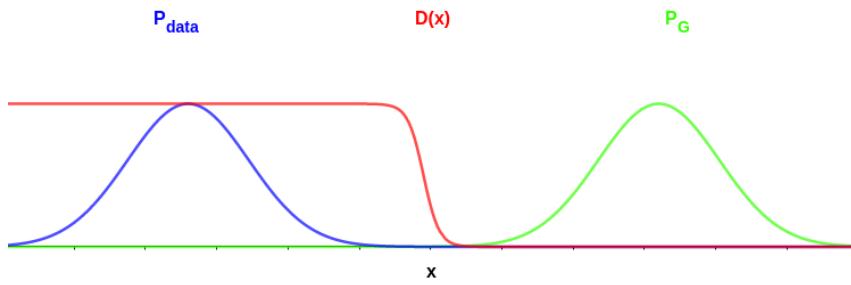


Figura 2.10: Esempio di *gradient vanishing*, causato da un discriminatore addestrato fino all’ottimo per un generatore non ottimo.

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:

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

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 ($KL(P_{data} \parallel P_G) \neq KL(P_G \parallel P_{data})$), 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*:

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

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 $W(P_{data}, P_G)$:

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

Analizzando le componenti di questa formula possiamo notare che il termine $\Pi(P_{data}, P_G)$ rappresenta l'insieme di tutte le possibili distribuzioni di probabilità γ che hanno come marginali le distribuzioni P_{data} e P_G , ovvero γ 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 γ 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 γ 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:

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

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:

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

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:

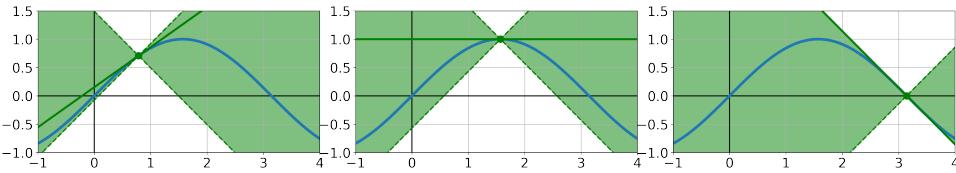


Figura 2.11: In questa immagine è possibile vedere la derivata della funzione seno, nei punti $\pi/4$, $\pi/2$ e π , ed è possibile vedere come tale funzione rispetti la *1-Lipschitz continuity*, essendo la sua derivata all'interno dell'intervallo ammesso in ogni punto.

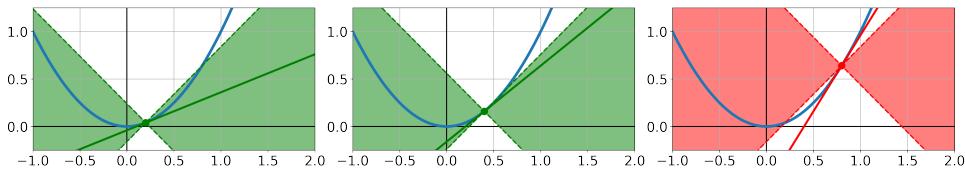


Figura 2.12: In questa immagine è possibile vedere la derivata della funzione x^2 , nei punti 0.2, 0.4 e 0.8, in questo caso la funzione non rispetta la *1-Lipschitz continuity*, in quanto la sua derivata cresce rapidamente oltre il limite consentito dopo $x=0.4$.

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.

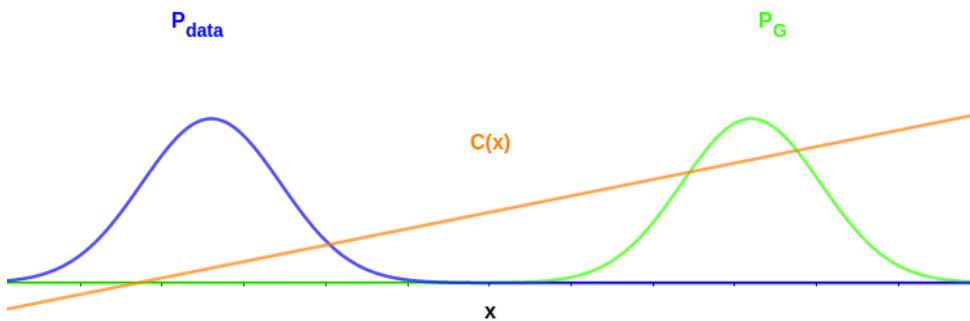


Figura 2.13: In questa immagine è possibile vedere come il critico $C(x)$ sia in grado di mappare efficacemente la distanza tra due distribuzioni P_{data} e P_G , restituendo dei gradienti utili per il generatore anche quando il critico è addestrato all'ottimo per il dato generatore.

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 efficace 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 so 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.

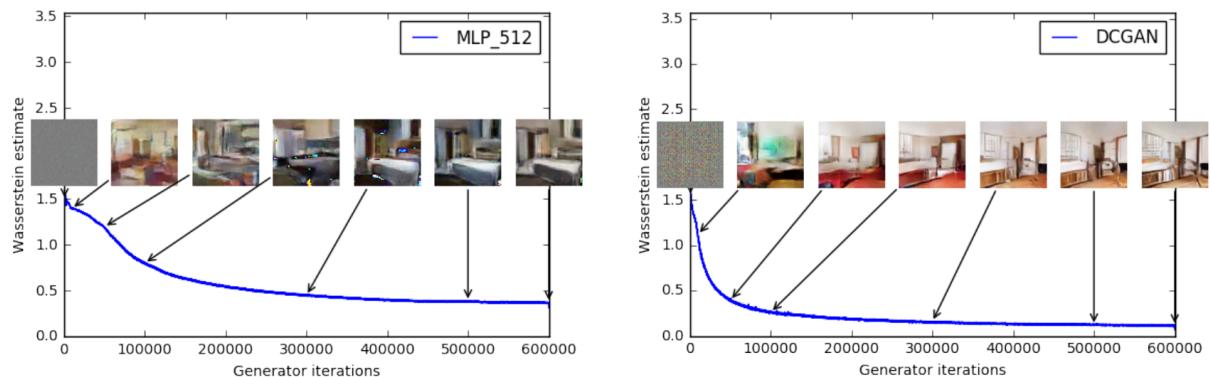


Figura 2.14: Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Wasserstein distance*. credits: Martin Arjovsky et al. [1]

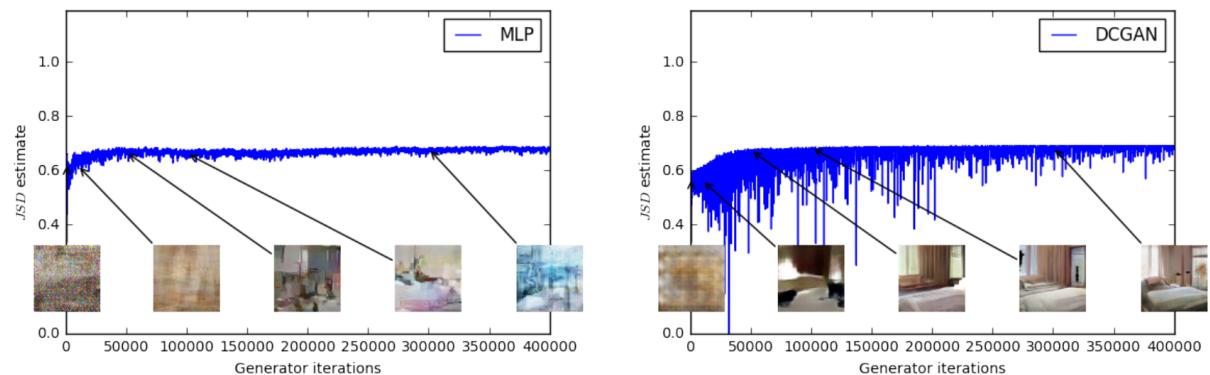


Figura 2.15: Andamento del training di una DCGAN e di una MLP addestrate utilizzando la *Jensen-Shannon divergence*. credits: Martin Arjovsky et al. [1]

2.4 Pix2Pix

Un’interessante lavoro presentato poco tempo dopo WGAN è stato Pix2Pix da parte di Phillip Isola et al. [13] 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*, il *super-resolution*, lo *style transfer*, il *denoising*, il *future frame prediction* e molti altri, tutti però hanno in comune il fatto che sono architetture o framework specializzati, realizzati appositamente per quel task, richiedendo competenze specifiche che hanno richiesto un accurato studio del problema e della soluzione. Pix2Pix invece ha proposto un’architettura che potesse 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.

2.4.1 La loss function

La *loss function* proposta da Pix2Pix è una variante della *loss function* proposta nella pubblicazione originale della GAN [9], 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 obiettivo, 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*:

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

Si noti che in questa equazione rispetto al caso classico precedentemente mostrato vanno identificate 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:

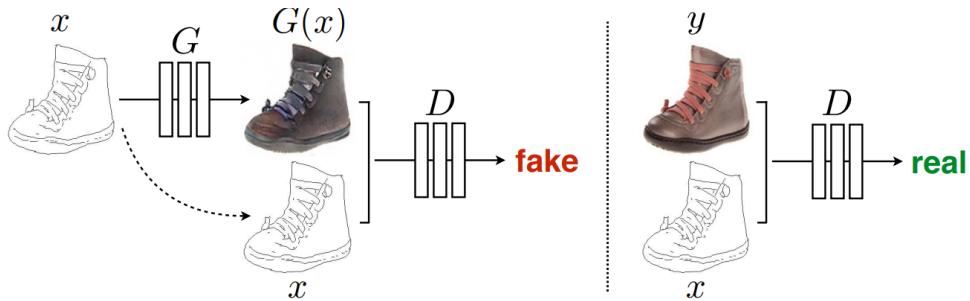


Figura 2.16: In questa immagine è possibile vedere un esempio di training di Pix2Pix per il task *edges to photo*. In tale configurazione il discriminatore apprende come discernere tra coppie di immagini e schizzi, e si può vedere come diversamente dal caso classico sia il generatore che il discriminatore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

credits: Phillip Isola et al. [13]

2.4.2 L'architettura

L'architettura proposta per Pix2Pix utilizza dei blocchi convoluzionali ripresi dalla precedentemente discussa DCGAN [20], 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 utilizzata con successo per il task di *image segmentation* da parte di Olaf Ronneberger et al. [21] nel 2015.

2.5 LAMA

TODO

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 è in continua crescita. 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'automatizzazione 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 Ševerstal 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>.

Questo dataset è diviso in 2 parti, training e test set, ma per il test set non sono stati rilasciati i ground truth, quindi non è possibile testare il modello sul test set originale e verranno dunque utilizzati solo i dati del training set, i quali verranno suddivisi 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 modello perfetto 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.

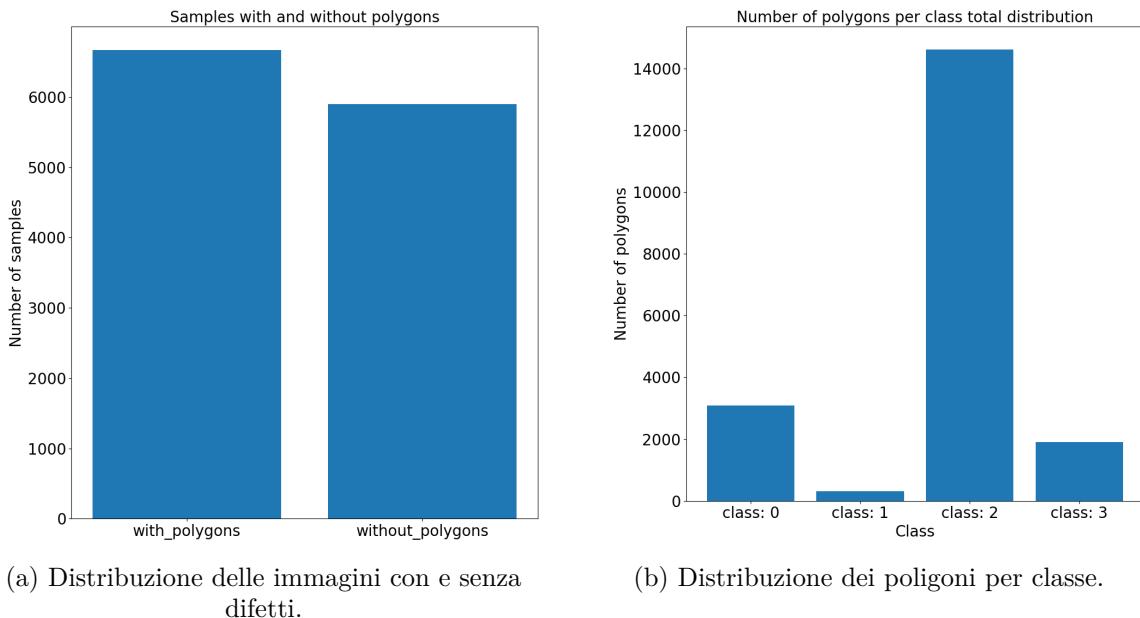
3.1.1 Distribuzione del dataset

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

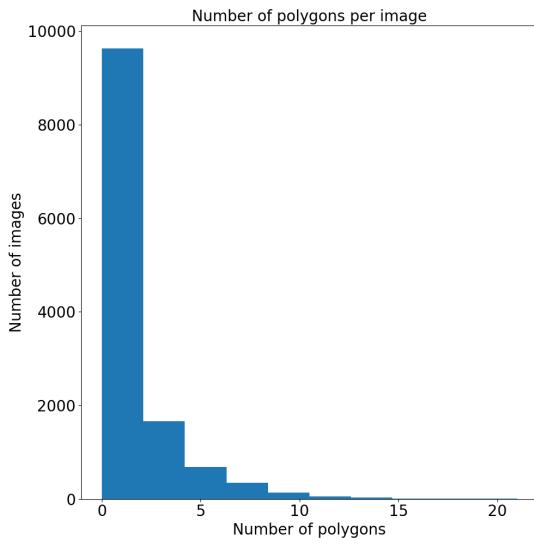
Tabella 3.1: Distribuzione delle immagini per classe.



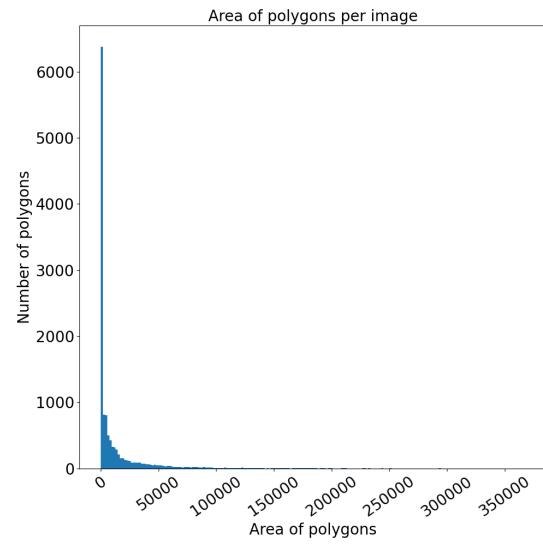
(a) Distribuzione delle immagini con e senza difetti.

(b) Distribuzione dei poligoni per classe.

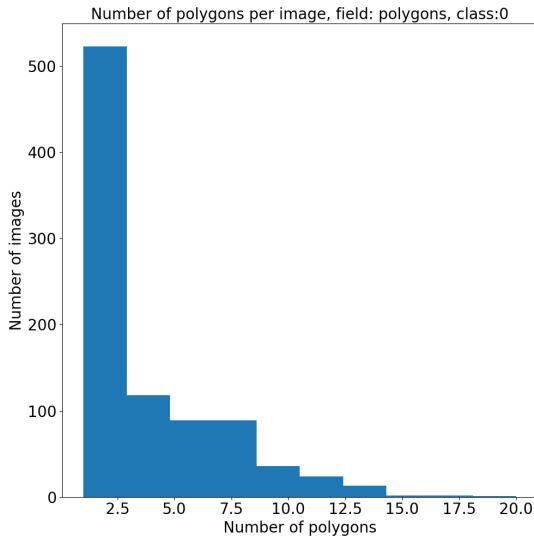
Altri parametri interessanti che vanno presi in considerazione riguardano la distribuzione dei poligoni nelle immagini, ovvero la distribuzione del numero di poligoni in ogni immagine, per avere un'idea della frequenza con cui questi sono presenti. Una valutazione è stata fatta anche per la distribuzione 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.



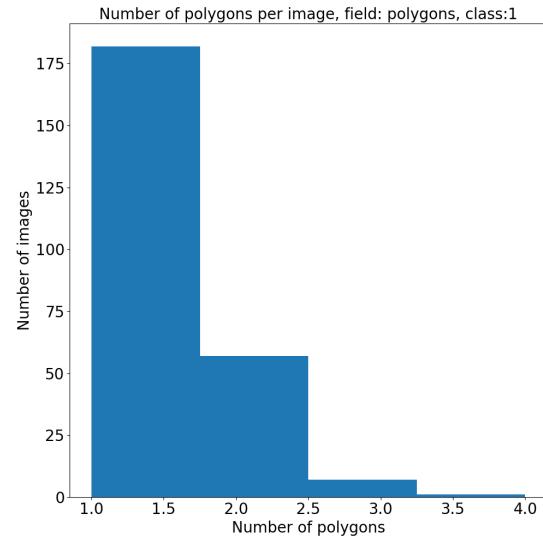
(a) Distribuzione numero poligoni per immagine.



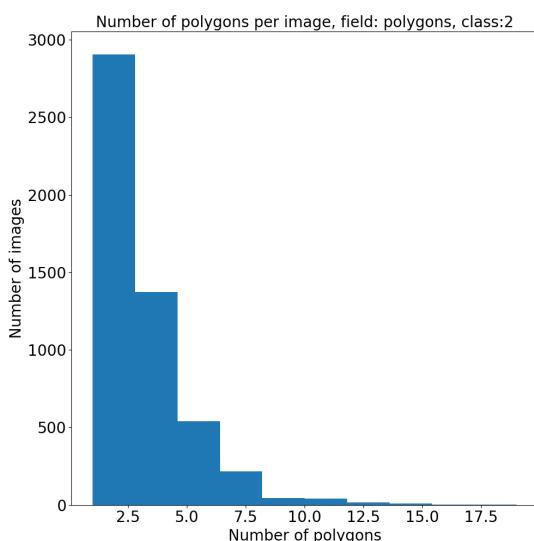
(b) Distribuzione area poligoni per immagine.



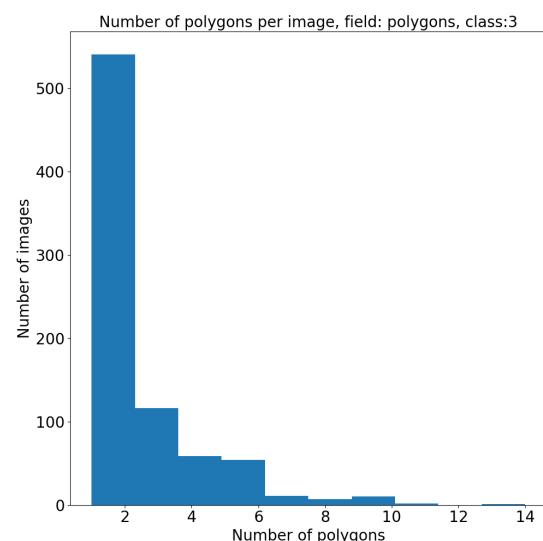
(c) Distribuzione numero poligoni classe 1 per immagine



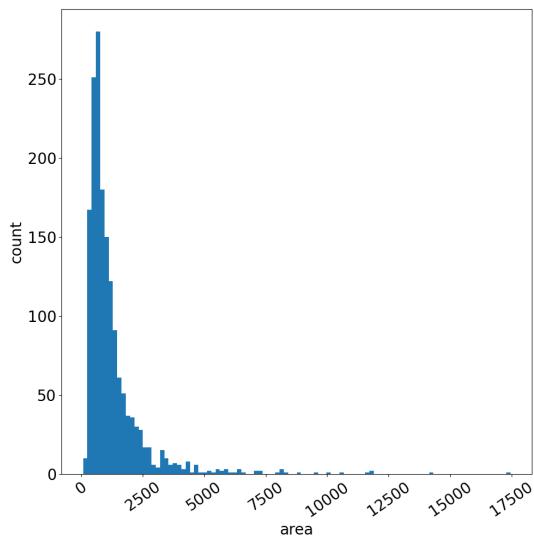
(d) Distribuzione numero poligoni classe 2 per immagine



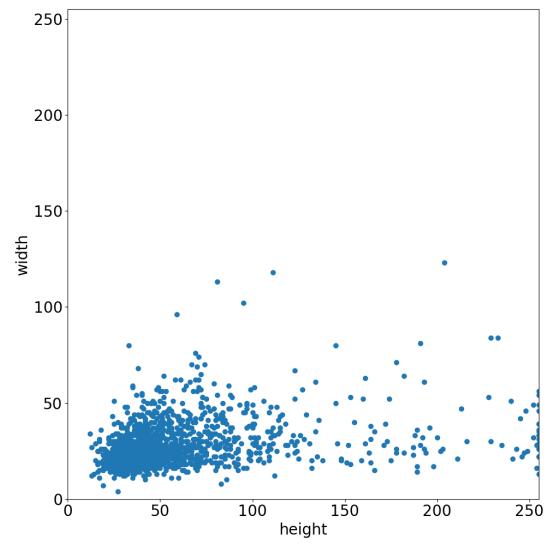
(e) Distribuzione numero poligoni classe 3 per immagine



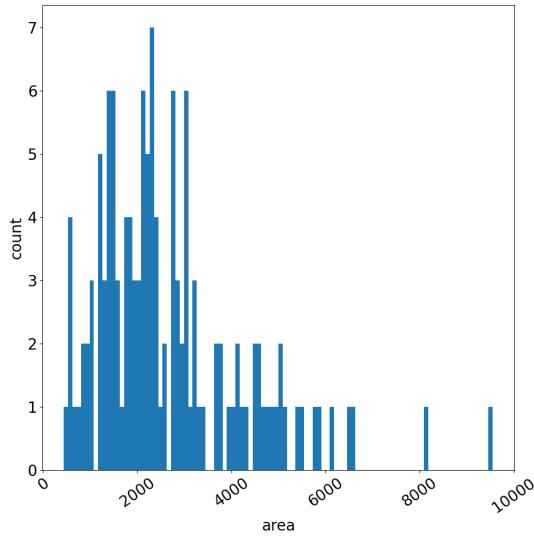
(f) Distribuzione numero poligoni classe 4 per immagine



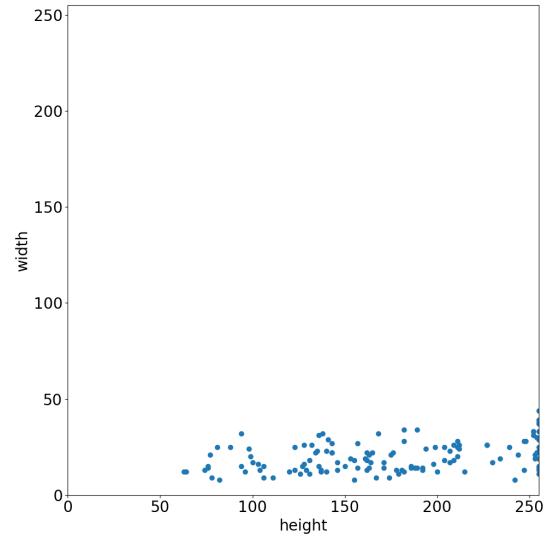
(a) Distribuzione area poligoni classe 1.



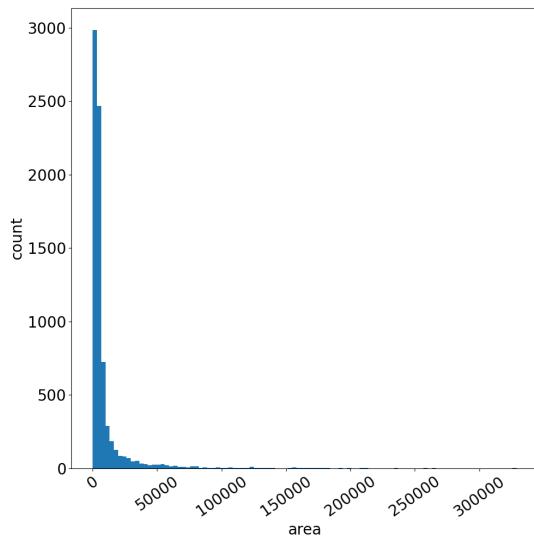
(b) Distribuzione aspect-ratio poligoni classe 1.



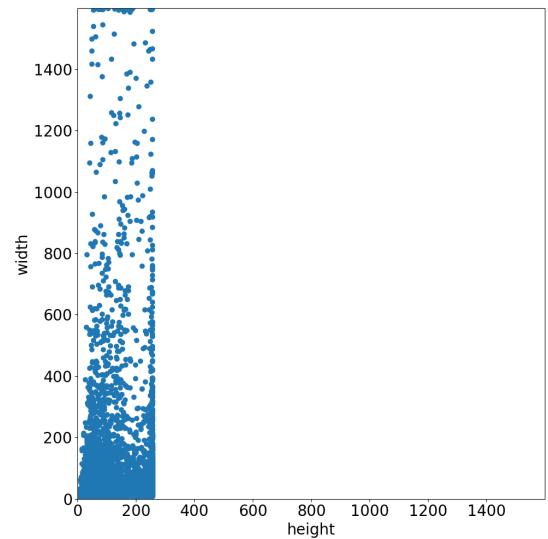
(a) Distribuzione area poligoni classe 2.



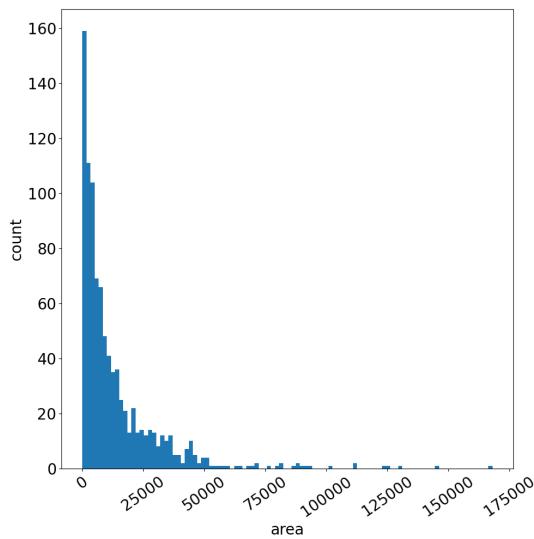
(b) Distribuzione aspect-ratio poligoni classe 2.



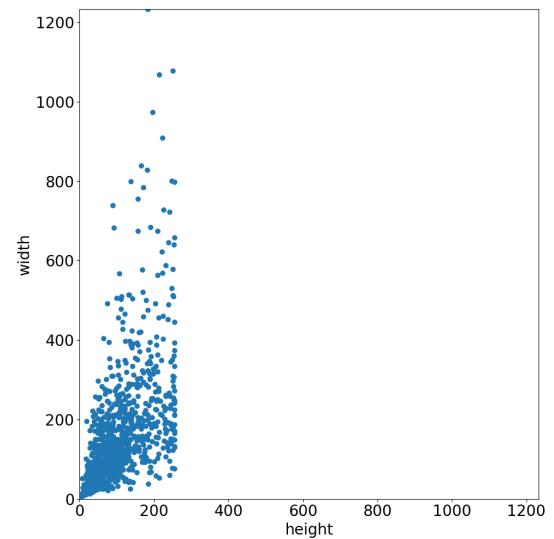
(a) Distribuzione area poligoni classe 3.



(b) Distribuzione aspect-ratio poligoni classe 3.

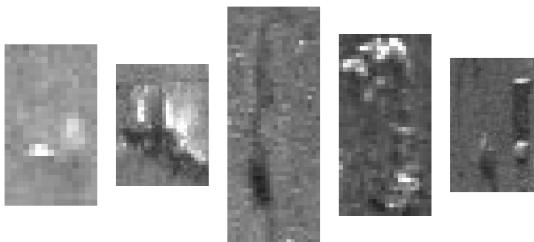


(a) Distribuzione area poligoni classe 4.

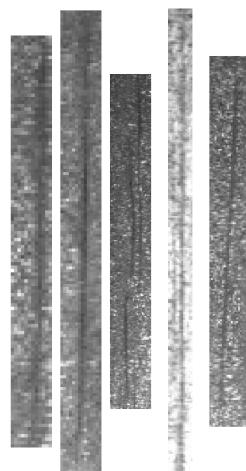


(b) Distribuzione aspect-ratio poligoni classe 4.

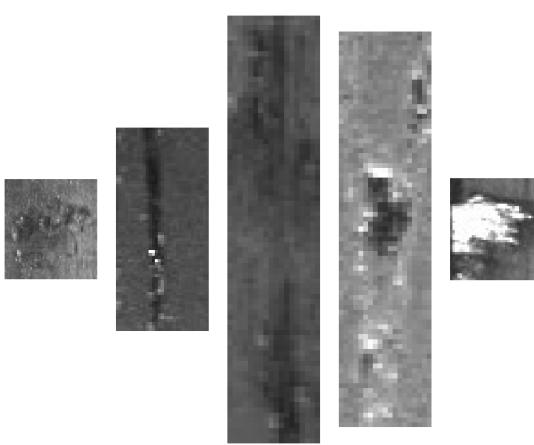
Vediamo di seguito alcuni esempi di difetti estratti dal dataset tramite crop, appartenenti alle quattro classi. Con il dataset non vengono fornite informazioni riguardo alle caratteristiche delle 4 classi ma è possibile dedurre alcune informazioni analizzando le immagini e le distribuzioni mostrate.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



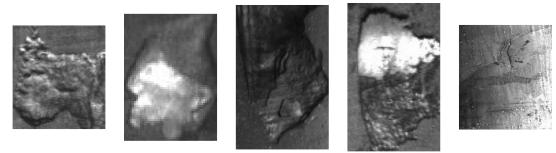
(a) Esempi di difetti classe 1.



(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.

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.

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.

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 retropropagazione 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

TODO

4.1 Definizione della pipeline di addestramento

TODO

4.2 Preparazione dei dati per la pipeline

TODO

4.3 Il dataloader

TODO

4.4 Il trainer

TODO

4.5 Lo script di evaluation

TODO

4.6 Tool per l'inferenza interattiva

TODO

Capitolo 5

Risultati

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 [5].	14
1.11	L'immagine tratta dallo stesso articolo illustra schematicamente l'architettura del modello TDNN. credits: Alex Waibel et al. 1989 [25].	14
1.12	L'immagine tratta dal medesimo articolo illustra l'architettura proposta. credits: Yann LeCun et al. 1989 [15].	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 [3].	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 [3]	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 [3]	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 [3]	21
1.19 Esempio di convoluzione di un’immagine rgb. credits: Irhum Shafkat [23]	22
1.20 Esempio di convoluzione multichannel. credits: Irhum Shafkat [23]	22
 2.1 Rappresentazione grafica della training pipeline di un modello GAN.	24
2.2 Confronto tra MSE e Adversarial training. credits: Goodfellow [8]	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. [9]	27
2.6 Esempio di corretto apprendimento (prima riga) e di mode collapse (seconda riga). credits: Luke Metz et al. [18]	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. [9]	29
2.8 Architettura del generatore di DCGAN. credits: Alec Radford et al. [20]	30
2.9 Esempio di algebra vettoriale nello spazio latente Z. credits: Alec Radford et al. [20]	31
2.10 Esempio di <i>gradient vanishing</i> , 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 π , 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 In questa immagine è mostrato l'andamento del training di una DCGAN e di una MLP (a 4 strati) addestrate utilizzando la *Wasserstein distance*, e alcune immagini generate durante l'addestramento. credits: Martin Arjovsky et al. [1] . 35
- 2.15 In questa immagine è mostrato l'andamento del training di una DCGAN e di una MLP (a 4 strati) addestrate utilizzando la *Jensen-Shannon divergence*, e alcune immagini generate durante l'addestramento. credits: Martin Arjovsky et al. [1] . 36
- 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. [13] 37

Bibliografia

- [1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.
- [2] George Cybenko. Approximation by superpositions of a sigmoidal function, 1989.
- [3] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.
- [4] 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.
- [5] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, 1980.
- [6] Rinon Gal, Dana Cohen, Amit Bermano, and Daniel Cohen-Or. Swagan: A style-based wavelet-driven generative model, 2021.
- [7] 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.
- [8] Ian J. Goodfellow. Introduction to generative adversarial networks, 2016.
- [9] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.
- [10] Anatoli Gorchev. Deep learning has reinvented quality control in manufacturing—but it hasn't gone far enough, 2020.
- [11] 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.
- [12] 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.
- [13] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.

- [14] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.
- [15] 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.
- [16] 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.
- [17] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge?, 2018.
- [18] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks, 2017.
- [19] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.
- [20] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.
- [21] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.
- [22] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.
- [23] Irhum Shafkat. Intuitively understanding convolutions for deep learning, 2018.
- [24] 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.
- [25] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. Phoneme recognition using time-delay neural networks, 1989.
- [26] 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.
- [27] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018.
- [28] Lili Zhu, Petros Spachos, Erica Pensini, and Konstantinos Plataniotis. Deep learning and machine vision for food processing: A survey, 2021.