

# UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA  
E MATEMATICA APPLICATA



Elaborato di Laurea in  
Ingegneria Informatica

## TECNICHE DI DATA AUGMENTATION BASATE SU CGAN PER IL SUPPORTO ALLA DIAGNOSI DEL PARKINSON

Relatore  
**Ch.mo Prof.**  
**Angelo Marcelli**

Candidato  
**Dylan Tangredi**  
**Mat. 0612705183**

Correlatore  
**Ing. Adolfo Santoro**

ANNO ACCADEMICO 2021/2022

“Now I’m a scientific expert;  
that means I know nothing about absolutely everything.”  
— *Arthur C. Clarke, 2001: A Space Odyssey*

# Abstract

La Malattia di Parkinson (MdP) è una patologia neurologica degenerativa a progressione lenta e la sua diagnosi è clinica, basata principalmente sulla cronologia dei sintomi e sulla visita neurologica. Alcuni degli esami supplementari necessari al supporto della diagnosi, però, possono risultare invasivi per il paziente. Inoltre, gli approcci diagnostici tradizionali potrebbero essere caratterizzati da soggettività, in quanto si basano sulla valutazione di sintomi motori spesso impercettibili e di conseguenza difficili da classificare, soprattutto nelle fasi iniziali della malattia, rendendo molto difficile una diagnosi precoce.

Per ridefinire e migliorare il processo diagnostico sono stati condotti diversi studi sul possibile utilizzo del Machine Learning (ML) come supporto alla diagnosi precoce della malattia, come per esempio l'utilizzo dei dati di scrittura a mano provenienti da persone affette da MdP e persone non affette. In questo caso si riscontra una scarsa disponibilità di campioni, dovuta alle difficoltà a cui il paziente va incontro durante il processo di raccolta dei dati, come affaticamento durante la scrittura o in generale problematiche dovute alle disabilità motorie.

L'obiettivo di questo lavoro è quello di valutare l'utilizzo di una Conditional Generative Adversarial Network (CGAN), una classe di reti neurali relativamente nuova del ML, per generare sinteticamente immagini relative ai dati di scrittura dei pazienti al fine di aumentarne la disponibilità, partendo da un dataset di immagini ottenute convertendo i segnali registrati da una smartpen; a tal fine si è reso necessario un lungo processo di trial and error sulla scelta e l'adattamento dei modelli e dei parametri di configurazione, oltre all'implementazione di uno script di ML completo, dal preprocessing dei dati al test del modello addestrato.

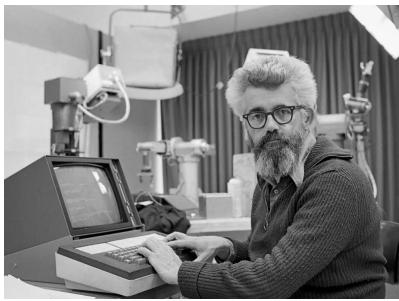
La valutazione finale raggiunta ci permette di indicare quali siano le difficoltà ancora da superare e di offrire uno spunto sulle possibili direzioni da esplorare in futuro.

# Indice

<b>Introduzione</b>	1
<b>1 Malattia di Parkinson e Machine Learning</b>	4
1.1 La Malattia di Parkinson . . . . .	4
1.2 IA per la diagnosi del Parkinson . . . . .	6
1.3 Machine Learning e Deep Learning . . . . .	6
1.3.1 Tipi di apprendimento . . . . .	8
1.3.2 Reti Neurali . . . . .	8
1.4 Modelli Generativi . . . . .	10
1.4.1 Generative Adversarial Networks (GAN) . . . . .	11
1.4.2 Conditional GAN (CGAN) . . . . .	12
<b>2 Sviluppo del modello CGAN</b>	14
2.1 Preprocessing . . . . .	14
2.1.1 Importazione delle librerie . . . . .	15
2.1.2 Acquisizione del dataset . . . . .	15
2.1.3 Normalizzazione dei dati . . . . .	18
2.2 Selezione ed adattamento dei modelli . . . . .	18
2.2.1 Generatore . . . . .	20
2.2.2 Discriminatore . . . . .	22
2.2.3 Definizione del modello combinato . . . . .	26
2.3 Ottimizzazione degli iperparametri . . . . .	27
2.4 Tracciamento dei progressi e dei risultati . . . . .	29
2.4.1 Callbacks API . . . . .	29
2.5 Addestramento della CGAN . . . . .	30
2.5.1 Test del modello addestrato . . . . .	32

<b>3 Valutazione dei risultati ed ulteriori esperimenti</b>	33
3.1 Ottimizzazione dei modelli . . . . .	33
3.2 Valutazione del modello finale . . . . .	35
3.2.1 Valutazione quantitativa . . . . .	35
3.2.2 Valutazione qualitativa . . . . .	42
3.3 Data augmentation . . . . .	43
3.3.1 Risultati del training con iperparametri di default . . . . .	44
3.3.2 Ulteriore tuning degli iperparametri . . . . .	44
3.4 Generazione condizionale delle immagini . . . . .	45
3.4.1 Valutazione qualitativa finale . . . . .	47
<b>Conclusioni</b>	48
<b>Riferimenti bibliografici</b>	50
<b>Elenco delle figure</b>	55
<b>Elenco delle tabelle</b>	57
<b>Elenco dei listati</b>	58

# Introduzione



**Figura 1:** John McCarthy nel suo laboratorio di IA a Stanford [1]

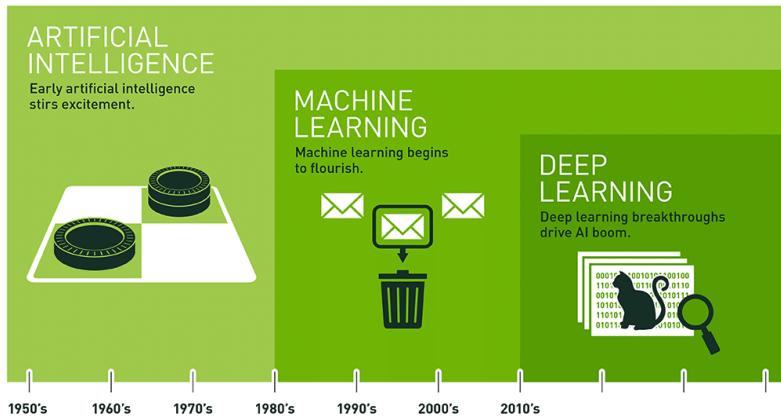
Nel corso dell'ultimo decennio, la locuzione Intelligenza Artificiale (IA) è entrata progressivamente a far parte di molte pubblicazioni scientifiche, anche in campo medico. Paradossalmente, però, l'IA è un concetto relativamente vecchio, che vede l'inizio della sua formalizzazione negli anni '40 e la nascita ufficiale del termine nel 1956 ad opera di John McCarthy [2]. In breve, IA si riferisce ad algoritmi informatici capaci di emulare caratteristiche tipiche dell'intelligenza umana, come l'apprendimento o il *problem solving*. I successi più recenti di questa tecnologia sono resi possibili dall'importante crescita della disponibilità di dati e della potenza di calcolo necessaria alla loro elaborazione. In particolare, le applicazioni di IA basate su algoritmi di Machine Learning (ML) e Deep Learning (DL) hanno avuto un impatto fondamentale nel campo della *computer vision*; la comunità medica ha approfittato di questi straordinari sviluppi per progettare applicazioni di IA che sfruttano al meglio le immagini di tipo medico, rendendo automatici differenti steps del processo clinico o per fornire supporto alle decisioni.

Molte pubblicazioni basate su IA mostrano risultati promettenti in una vasta gamma di applicazioni mediche [3][4][5]; la diagnosi delle patologie, la segmentazione delle immagini o la previsione dei risultati rappresentano solo alcune delle attività soggette a grandi trasformazioni grazie ai progressi di questa tecnologia.

Più recentemente, gli strumenti di ML sono divenuti abbastanza maturi da poter soddisfare a pieno i requisiti clinici; di conseguenza, gruppi di ricerca e di professionisti in campo medico lavorano con le aziende specializzate con l'obiettivo comune di sviluppare soluzioni di IA da applicare ai diversi settori medico-sanitari.

---

Oggi siamo molto vicini all'implementazione clinica dell'IA e ciò rende la conoscenza delle basi di questa tecnologia necessaria per ogni professionista nel settore medico; permettere alla comunità medica di acquisire un solida conoscenza di *background* su IA e metodi di apprendimento, inclusi la loro evoluzione e lo stato dell'arte corrente, risulterà in una ricerca di qualità più alta, in un'agevolazione per i nuovi ricercatori nel campo e in un'ispirazione maggiore verso nuove direzioni di ricerca.



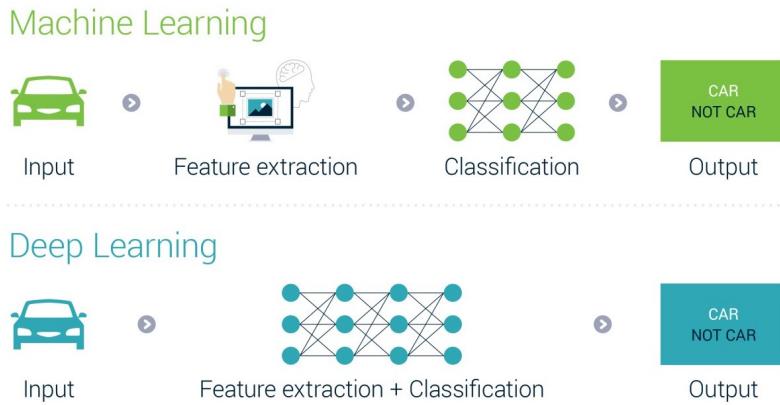
**Figura 2:** Intelligenza Artificiale - *timeline* [6]

In questo lavoro è mostrato un particolare utilizzo di tecnologie di ML per supportare la diagnosi della Malattia di Parkinson (MdP); la MdP è una patologia che richiede esami spesso invadenti ma soprattutto difficili da sostenere per i pazienti, e ciò causa anche una carenza di dati associabili alla MdP con conseguente maggiore difficoltà nell'esprimere una diagnosi accurata.

Nello specifico, si è voluto dimostrare come l'utilizzo del ML può essere utile per classificare i dati provenienti dagli esami clinici classici a cui le persone affette da MdP sono sottoposte, in particolare da esercizi di scrittura a mano, ed eventualmente generare nuovi dati sintetici per ovviare alla carenza di dati originali. Partendo dai segnali acquisiti da una smartpen [7] e convertiti in immagini [8], il nostro obiettivo è stato quello di utilizzare una particolare architettura, la *Conditional Generative Adversarial Networks* (CGAN), formata da due reti neurali (RN) in opposizione tra loro e addestrarla sulle immagini, che compongono il *dataset* di partenza; l'uso di algoritmi di DL, un sottoinsieme del ML, ci permette di distinguere e classificare i dati provenienti da persone affette da MdP e persone non affette,

---

in base alle caratteristiche presenti nelle immagini convertite dai segnali catturati dalla smartpen.



**Figura 3:** Machine Learning vs Deep Learning [9]

Una volta completato il *training* delle RN, l'obiettivo finale è stato quello di utilizzare il modello addestrato per generare nuove immagini sintetiche, arricchendo il dataset iniziale grazie al processo di *data augmentation*.

L'elaborato è organizzato come segue:

Il **Capitolo 1** contiene informazioni sulla MdP e sul ML: è introdotta in breve la patologia e sono mostrati i concetti alla base del ML e della sua particolare applicazione su cui si basa questo lavoro, l'architettura CGAN.

Il **Capitolo 2** presenta in dettaglio la parte operativa del lavoro; è mostrata l'intera *pipeline* dell'applicazione di ML sviluppata, dalla fase di *preprocessing* al training e test della RN, insieme alle motivazioni per ogni scelta fatta in fase di ottimizzazione dei modelli.

Il **Capitolo 3** si può considerare diviso in due parti: nella prima sono presentati i risultati relativi al processo di sviluppo dell'architettura scelta e la valutazione finale, sia quantitativa che qualitativa, del modello addestrato. Nella seconda parte sono mostrati gli ulteriori esperimenti effettuati per verificare e dimostrare i risultati ottenuti in precedenza, con il confronto finale degli stessi in forma di valutazione qualitativa delle immagini sintetiche.

Nelle **Conclusioni** sono raccolti sia i motivi dietro al lavoro svolto che un resoconto del processo di sviluppo e dei risultati ottenuti nelle varie fasi, oltre alle considerazioni su possibili sviluppi futuri.

# **Capitolo 1**

## **Malattia di Parkinson e Machine Learning**

Negli ultimi anni, il Machine Learning ha iniziato a svolgere un ruolo fondamentale in ambito medico e sanitario. Come si evince dal nome, che si può tradurre con "apprendimento automatico", il Machine Learning permette ad un sistema informatico di apprendere ed estrarre delle rappresentazioni significative dai dati in maniera automatica o semi-automatica.

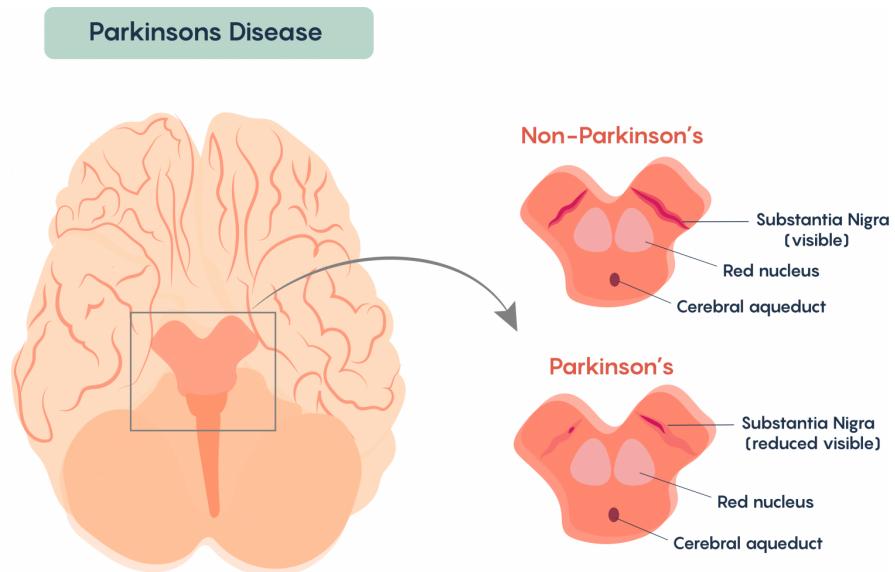
In questo capitolo introduciamo come l'Intelligenza Artificiale viene utilizzata in ambito medico e sanitario, ponendo particolare attenzione al Machine Learning e al Deep Learning e a come sono attualmente utilizzati per favorire una diagnosi tempestiva ed accurata della Malattia di Parkinson; inoltre, verranno introdotte le tecnologie e i modelli di Machine Learning e Reti Neurali sulle quali si basa il lavoro di questo elaborato per favorire la comprensione dei contenuti presenti nei capitoli successivi, focalizzati principalmente sulla parte operativa dello studio.

### **1.1 La Malattia di Parkinson**

La Malattia di Parkinson (MdP) è una malattia neurodegenerativa che agisce principalmente sui neuroni che producono dopamina, in una regione specifica del cervello detta "substantia nigra", o sostanza nera di Sommering. I sintomi si sviluppano gradualmente nel corso degli anni e la progressione dei suddetti sintomi spesso varia da persona a persona a causa della natura della patologia. Le persone affette da

MdP possono essere soggette a tremore, bradicinesia, rigidità muscolare, problemi nella deambulazione e nell'equilibrio. [10]

La MdP si manifesta nel momento in cui le cellule nervose (neuroni) nella sostanza nera di Sommerring sono già in buona parte compromesse. La produzione di dopamina, un neurotrasmettore che aiuta la comunicazione tra le cellule cerebrali, viene così limitata. La dopamina è fondamentale anche per le operazioni svolte in un'altra area del cervello, costituita dai cosiddetti gangli della base, responsabili dell'organizzazione dei comandi cerebrali necessari per un corretto movimento del corpo; da questo fenomeno, quindi, nascono i sintomi elencati precedentemente, tipici di un paziente con malattia di Parkinson. [11]



**Figura 1.1:** Perdita dei neuroni dopaminerigici della substantia nigra [12]

Più recentemente, studi hanno dimostrato che, oltre alla dopamina, un altro neurotrasmettore gioca un ruolo fondamentale nello sviluppo della malattia: la noradrenalina (NA). Anche la NA è coinvolta in una grande varietà di funzioni cognitive e motorie del cervello; dai test sperimentali si evince come il declino della sua sintesi potrebbe anch'esso essere un fattore importante nel processo degenerativo della MdP, che spiegherebbe la presenza di sintomi non motori come, per esempio, disturbi del sonno e depressione. [13]

Ancora oggi, non sono esattamente note le cause che portano alla degenerazione delle cellule nervose causanti la MdP; la ricerca per identificare le potenziali cause

è tutt’oggi in corso. Al momento, gli studi evidenziano come la responsabilità sia da attribuire ad una combinazione di cambiamenti genetici e fattori ambientali. Infatti, è stato mostrato come specifici tratti genetici possano portare ad un aumento del rischio di contrarre la malattia, nonostante non sia chiaro come essi rendano il soggetto più suscettibile ad essa. La MdP può essere ereditaria, come risultato del passaggio di un gene da genitore a figlio, pur essendo un’evenienza rara. Alcuni ricercatori, invece, suggeriscono come alcuni pesticidi e diserbanti usati in agricoltura, oltre all’inquinamento industriale, potrebbero contribuire alla nascita della patologia; le prove, però, risultano al momento inconclusive. [14]

## 1.2 IA per la diagnosi del Parkinson

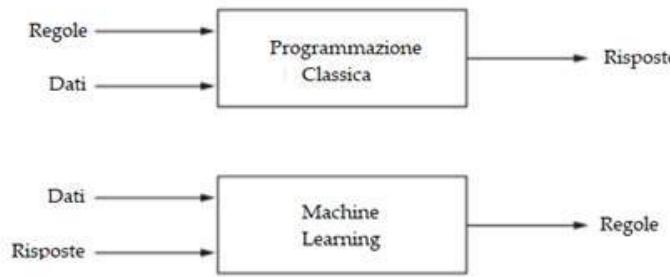
Recentemente, anche nel settore medico è aumentato l’utilizzo dell’Intelligenza Artificiale e delle tecniche di ML. L’IA nasce insieme ai primi computer e, nel corso degli anni, ha alternato diversi periodi di grande interesse, dovuti ai particolari successi ottenuti, con periodi in cui si è perso interesse per la materia a causa dei fallimenti sopraggiunti. Il limite più grande incontrato durante gli anni è stato l’insufficiente potenza di elaborazione; ad oggi, però, questo problema può considerarsi superato grazie alla disponibilità di hardware sempre più performante e al supporto di tecnologie nuove [15].

Nella diagnosi della MdP, i modelli di ML sono applicati a molteplici tipi di dati, provenienti per esempio da attività motoria [16] e vocale [17], imaging cerebrale [18] e da scrittura a mano [7]. L’utilizzo di un approccio basato su tecniche di ML rende possibile l’identificazione di *features* rilevanti che non sono tradizionalmente usate nella diagnosi clinica della MdP e che fanno affidamento su questo tipo di tecniche per scoprire la presenza della malattia nella fase preclinica o in una sua forma atipica.

## 1.3 Machine Learning e Deep Learning

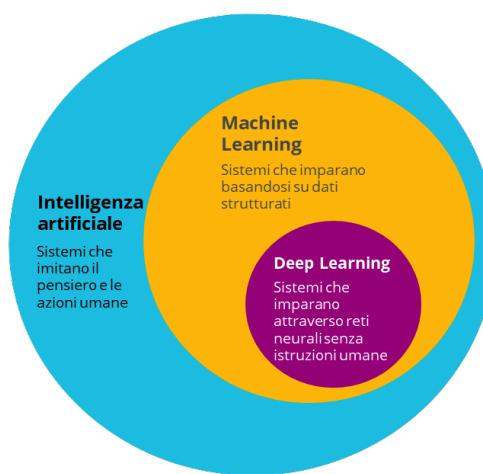
Il ML è un sottoinsieme dell’Intelligenza Artificiale e, insieme al Deep Learning (DL), costituisce un approccio moderno che sta riscuotendo un notevole successo anche in campo medico. Principalmente, il ML si occupa di fare delle previsioni, “imparando” dai dati. Partendo da un dataset, il data scientist implementa un

algoritmo capace di restituire la previsione desiderata. Il campo di applicazione del ML può essere enorme: è possibile applicarlo a diversi problemi e la soluzione viene proposta direttamente dall'algoritmo. Al contrario degli approcci informatici classici, non si tratta di sviluppare un programma che propone un risultato a partire dai dati, ma in maniera opposta si parte dai risultati per ottenere delle regole [15].



**Figura 1.2:** Programmazione classica vs Machine Learning [15]

Infine, il DL è un sottoinsieme del ML ed entra in gioco quando vi è a disposizione un discreto volume di dati da utilizzare per far apprendere i modelli; esso riesce ad elaborare grosse moli di dati e ad ottenere risultati più precisi, grazie a modelli di reti neurali particolarmente complessi ed organizzati gerarchicamente, al costo di una potenza di calcolo maggiore e tempi di elaborazione più lunghi.



**Figura 1.3:** Intelligenza Artificiale, Machine Learning e Deep Learning [19]

### 1.3.1 Tipi di apprendimento

Gli algoritmi di ML possono essere raggruppati in quattro grandi famiglie [15]:

**Apprendimento Supervisionato** (Supervised Learning): gli algoritmi di questo tipo utilizzano dataset in cui è già presente la "risposta giusta", imparando da essi a prevedere la risposta per un nuovo insieme di dati, che non fa parte dell'insieme utilizzato per l'addestramento.

**Apprendimento Non Supervisionato** (Unsupervised Learning): questi algoritmi usano dataset che non hanno delle risposte disponibili. Analizzando i dati, l'algoritmo stesso produce delle relazioni tra gli stessi e genera le possibili risposte, aiutando a scoprire nuove informazioni presenti nei dati.

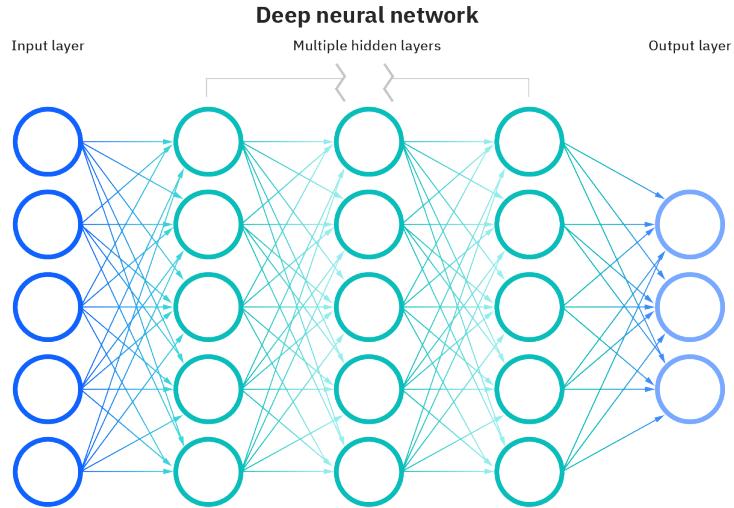
**Apprendimento Semi Supervisionato** (Semi-Supervised Learning): a metà tra i due precedenti, questo tipo di algoritmo utilizza dataset in cui la risposta è presente solo per una piccola porzione degli esempi; l'obiettivo sarà quello di ottenere una regola generale da sfruttare per i dati senza risposta, provando ad utilizzare al meglio i pochi dati che hanno una risposta disponibile.

**Apprendimento per rinforzo** (Reinforcement Learning): questi algoritmi non prevedono un dataset di training e di conseguenza non hanno un'esperienza da utilizzare per imparare; cercano di arrivare al risultato ottimale partendo da una situazione iniziale qualsiasi, attraverso regole di premiazione o penalizzazione. Un tipico esempio è quello di un videogioco: un algoritmo può imparare a giocare grazie alle ricompense ottenute ogni volta che esegue la mossa giusta e alle penalizzazioni ottenute quando esegue una mossa sbagliata.

### 1.3.2 Reti Neurali

Il cuore degli algoritmi di DL è rappresentato dalle Reti Neurali (RN), o (*Neural Networks*) [20]. Il loro nome e la loro struttura sono ispirati dal cervello umano, imitando il modo in cui i neuroni biologici comunicano tramite l'invio reciproco di segnali. Le RN artificiali sono costituite da strati di nodi o *node layers*, contenenti un *input layer*, uno o più strati nascosti (*hidden layers*) e un *output layer*. Ciascuno dei nodi, o neuroni artificiali, si connette ad un altro nodo ed ha un peso (*weight*), ed una soglia (*threshold*) associati. Se l'output di qualunque nodo individuale si trova al di sopra del valore di *threshold* specificato, quel nodo viene attivato inviando

dati al *layer* successivo della rete; in caso contrario, nessun dato viene passato al successivo *layer*.



**Figura 1.4:** Esempio di Deep Neural Network [20]

Le RN si affidano ai dati di addestramento *training data* per apprendere e migliorare la loro precisione, o *accuracy* con il tempo; una volta che gli algoritmi di apprendimento sono ottimizzati per l'*accuracy* rappresentano dei potenti mezzi per la *computer science* e l'IA, permettendo di raccogliere e classificare dati con estrema velocità. Attività come il riconoscimento di immagini, per esempio, possono impiegare minuti contro le ore necessarie per l'identificazione manuale da parte dell'uomo.

### Tipi di Rete Neurale

Le RN si possono classificare in diversi tipi, utilizzati per scopi diversi. Tra i tipi più comuni è importante citare le *Recurrent Neural Networks (RNN)*, utilizzate principalmente quando si lavora su serie temporali per effettuare previsioni su risultati futuri; da una RNN nasce, per esempio, il dataset di immagini utilizzate in questo lavoro [8], presentato in dettaglio nella sezione 2.1.2.

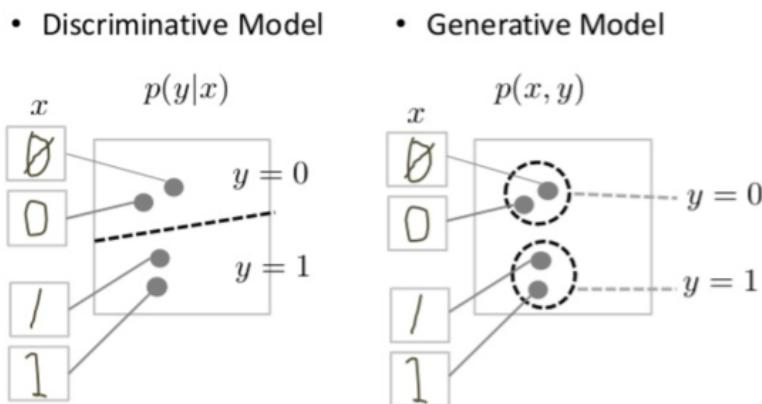
Un altro importante tipo di RN sono le *Convolutional Neural Networks (CNN)*, particolarmente adatte alle attività relative al riconoscimento delle immagini, dei *pattern*, o alla *computer vision*. Le CNN si basano su principi di algebra lineare,

soprattutto su operazioni tra matrici, per identificare eventuali *pattern* presenti all'interno di un'immagine. Come vedremo nel 2.2, il lavoro di questo elaborato si basa proprio su un tipo particolare di CNN.

## 1.4 Modelli Generativi

La Rete Neurale Convolutzionale (CNN) presentata nel 2.2 è una RN costituita da molteplici *hidden layers* addestrati per approssimare complicate distribuzioni di probabilità multi-dimensionali [21]. Questo tipo di RN fa parte dei Modelli Generativi, una classe di modelli statistici a cui si contrappongono, in generale, i Modelli Discriminativi.

In modo informale, i modelli generativi sono capaci di generare nuove istanze di dati, mentre i modelli discriminativi possono distinguere i diversi tipi di istanze di dati. Per esempio, un modello generativo potrebbe generare nuove immagini di animali che sembrano a tutti gli effetti animali reali, mentre il modello discriminativo potrebbe distinguere l'immagine di un cane da quella di un gatto [22]. Esposto in maniera più formale, dato un set di istanze di dati  $x$  e un set di etichette  $y$ , i modelli generativi catturano la probabilità congiunta  $p(x,y)$ , o solamente  $p(x)$  se non sono presenti etichette; i modelli discriminativi, invece, catturano la probabilità condizionata  $p(y/x)$

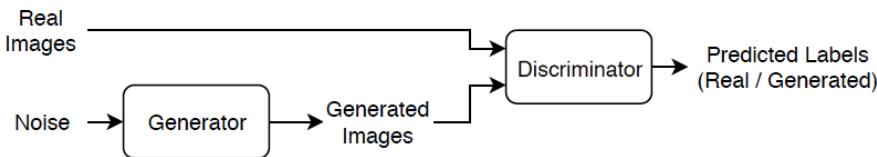


**Figura 1.5:** Modelli discriminativi e generativi su cifre scritte a mano [22]

### 1.4.1 Generative Adversarial Networks (GAN)

Nel 2014, Goodfellow et al. [23] proposero una nuova architettura, chiamata Generative Adversarial Networks (GAN), per la stima dei modelli generativi attraverso un processo "avversario", nel quale vengono simultaneamente addestrati due diversi modelli: un modello generativo  $G$  che cattura la distribuzione dei dati e un modello discriminativo  $D$  che stima la probabilità che un campione provenga dai dati di training piuttosto che dal modello  $G$ . La procedura di addestramento di  $G$  ha lo scopo di massimizzare la probabilità che  $D$  commetta un errore. L'architettura così descritta corrisponde ad un cosiddetto gioco *minimax* tra le due entità, in cui il modello generativo  $G$  si scontra con un avversario: il modello discriminatore  $D$ .

La figura 1.6 mostra il modello dell'architettura GAN classica.



**Figura 1.6:** Modello della Generative Adversarial Network (GAN) [24]

Come analogia, l'autore stesso propone di pensare al modello generativo  $G$  come ad un falsario che tenta di produrre banconote contraffatte, mentre il modello discriminativo  $D$  gioca il ruolo delle autorità che cercano di individuare le banconote false. La competizione in quest'attività spinge entrambe le parti a migliorare, fino ad arrivare al punto in cui le banconote contraffatte risultano indistinguibili dalle originali.

All'inizio dell'addestramento, il generatore produce dei dati palesemente falsi e il discriminatore li riconosce facilmente [22]:



**Figura 1.7:** Generatore vs Discriminatore - Inizio dell'addestramento

Con l'avanzare dell'addestramento, il generatore si avvicina alla produzione di un output capace di ingannare il discriminatore:



**Figura 1.8:** Generatore vs Discriminatore - Progresso dell’addestramento

Se l’addestramento del generatore va a buon fine, il discriminatore inizia a commettere degli errori di classificazione; non essendo più così abile a distinguere con certezza i campioni sintetici da quelli reali, la sua *accuracy* decresce.



**Figura 1.9:** Generatore vs Discriminatore - Fine dell’addestramento

### 1.4.2 Conditional GAN (CGAN)

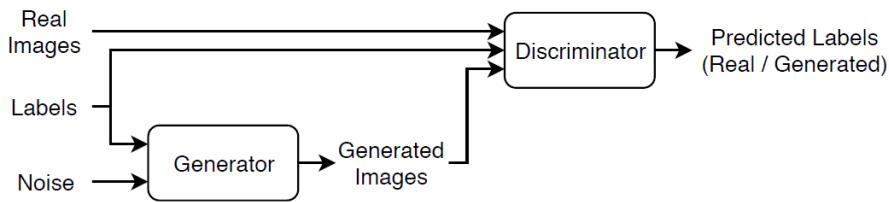
Tra le tante implementazioni dell’architettura GAN ideata da Ian Goodfellow [23], per questo lavoro si è scelto di utilizzare una *Conditional GAN* (CGAN), presentata per la prima volta da Mirza et al. alle fine del 2014 [25]. Si tratta delle versione condizionale della GAN, costruita semplicemente fornendo sia al generatore che al discriminatore i dati  $y$  sui quali condizionare. In questo modo, a differenza dei modelli generativi non condizionati, è possibile guidare il processo di generazione dei dati sintetici grazie alle informazioni addizionali fornite ai modelli; il condizionamento può essere basato su diversi oggetti, tra cui le etichette delle classi, o *class labels*, variante scelta per questo lavoro. I vantaggi nell’utilizzo di una CGAN sono principalmente due [26]:

- Convergenza più rapida - anche la distribuzione casuale che le immagini sintetiche seguono avranno del *pattern*, dando la possibilità di verificare in tempi relativamente veloci se il training procede verso la direzione desiderata.
- Controllo sull’output - alla fine dell’addestramento sarà possibile scegliere il tipo di immagine da generare indicando l’etichetta della classe. Ad esempio,

avranno la possibilità di generare immagini sintetiche per un dataset originalmente sbilanciato, cioè che presenta un numero minore di campioni per una classe rispetto ad un'altra, aumentando effettivamente il numero di campioni totali per il training; il processo appena descritto prende il nome di *data augmentation* e coincide con l'obiettivo finale di questo elaborato.

La figura 1.10 mostra un modello di architettura CGAN:

- Generatore - prende in input un'etichetta e un vettore di valori casuali dallo spazio latente, cioè rumore o *noise*. In output  $G$  produce dati sintetici che hanno la stessa struttura delle osservazioni dei dati di training corrispondenti alla stessa etichetta, nel nostro caso immagini.
- Discriminatore - prende in input dei *batch* di dati  $x$  contenenti osservazioni provenienti sia dai dati di training che da quelli generati da  $G$ , nel nostro caso immagini reali e sintetiche, e le etichette relative corrispondente alle loro classi di appartenenza  $y$ ; il suo output è la previsione binaria che le osservazioni siano reali o sintetiche.



**Figura 1.10:** Conditional Generative Adversarial Network (CGAN) [25]

Nel capitolo 2 verrà presentata l'implementazione della CGAN utilizzata in questo elaborato. L'obiettivo finale, come anticipato, sarà quello di aggiungere immagini sintetiche al dataset originale per ovviare alla scarsa disponibilità di campioni utilizzando tecniche di *data augmentation*.

# Capitolo 2

## Sviluppo del modello CGAN

In questo capitolo è presentato in dettaglio il lavoro svolto per implementare una Conditional Generative Adversarial Network (CGAN) che lavori sul dataset di immagini ottenuto dalla conversione di segnali temporali estratti da dati di scrittura di persone affette da morbo di Parkinson e dal gruppo di controllo. In particolare, sono mostrate in dettaglio tutte le fasi della pipeline che, generalmente, costituiscono un'applicazione di Machine Learning, dal preprocessing dei dati alla generazione delle immagini sintetiche.

L'obiettivo finale è stato quello di addestrare la rete CGAN sul dataset e aggiungere le immagini sintetiche generate dal modello al dataset iniziale, aumentando effettivamente il numero di campioni disponibili per l'addestramento. Come riferimento è stato considerato lo script in linguaggio Python [27] di Sayak Paul "Conditional GAN" [28], originariamente costruito per l'utilizzo sul dataset pubblico MNIST [29], un database formato da 70,000 immagini rappresentanti cifre scritte a mano; nei paragrafi successivi modificheremo lo script per adattarlo al nostro dataset.

### 2.1 Preprocessing

Una porzione importante di ogni progetto basato su dati è costituita dal preprocessing degli stessi. In generale, esso consiste nell'analisi, filtraggio, trasformazione e codifica dei dati affinché un algoritmo di Machine Learning possa comprenderli e lavorare sull'output del processo. La fase di preprocessing della pipeline è composta da varie sezioni, che esploreremo in dettaglio.

### 2.1.1 Importazione delle librerie

La fase di preprocessing inizia con l'importazione delle librerie e dei moduli necessari al funzionamento di qualunque script in linguaggio Python. Nel nostro caso, l'applicazione di Machine Learning da sviluppare richiede librerie specializzate, come il framework Tensorflow [30] su cui si basa il nostro lavoro e la dashboard Wandb [31] che facilita il tracciamento degli esperimenti.

Altre librerie essenziali per la ogni computazione scientifica, incluso il Machine Learning, sono NumPy [32] e Pandas [33]; il primo è un modulo open source che consente di effettuare operazioni matematiche su array e matrici in modo efficiente, la seconda offre strutture dati e strumenti di analisi ad alte prestazioni, indispensabili nel campo della *data science*. Non mancano inoltre librerie di supporto per la realizzazione di grafici e la gestione del file system come, per esempio, Matplotlib [34] e pathlib [27].

```
1 import tensorflow as tf
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import os
5 import glob
6 import pathlib
7 import pandas as pd
8 import time
9 import wandb
10 from wandb.keras import WandbCallback
```

**Listato 2.1:** Importazione delle librerie necessarie

### 2.1.2 Acquisizione del dataset

Il dataset utilizzato per l'attività è costituito da immagini .png di risoluzione di 64x64, in scala di grigi. Si basa sul dataset NewHandPD di Pereira et al. [7], con i dati di 66 persone divise in due gruppi: 31 persone affette da Parkinson e 35 non affette. I dati utilizzati per questa attività riguardano solo i tasks "meander" e "spiral", esercizi di scrittura in cui il soggetto utilizza una smartpen che memorizza i segnali temporali tramite i sensori di cui è equipaggiata. I segnali sono stati convertiti in immagini prendendo come riferimento la tesi di laurea magistrale dell'Ing. Gennaro Gemito [8] e successivamente suddivisi e bilanciati in 5 seeds ognuno, ottenendo circa 120 campioni (60 per classe) per il training e 120 per il test su ogni seed.

Tensorflow [30] fornisce il metodo adatto al caricamento del dataset da una directory del file system, *image\_dataset\_from\_directory()*, che restituisce gruppi

di immagini dalle sottocartelle corrispondenti alle classi, comprensive di etichette 0 e 1 (corrispondenti rispettivamente alle classi Healthy e Patient). I parametri del metodo permettono di specificare la *directory* contenente il dataset, sia per i campioni riservati a train e validation, sia per i campioni riservati al test.

Per i parametri relativi alle etichette, *labels* e *label\_mode*, abbiamo specificato rispettivamente i valori *inferred* e *categorical*, tramite i quali è possibile generare le etichette direttamente dalla struttura delle sottocartelle e codificarle come un vettore di categorie. Il formato delle immagini del dataset è .png, e il metodo ci permette di specificare la risoluzione alla quale ridimensionare le stesse dopo il caricamento dal disco, con *image\_size*, e il numero di canali corrispondenti al *color\_mode*, nel nostro caso "1" per immagini *grayscale*. Da notare che il parametro relativo alla risoluzione è obbligatorio, in quanto la pipeline processa immagini che devono avere le stesse dimensioni. Infine, possiamo impostare il *batch\_size*, di cui parleremo nel dettaglio nella sezione riguardante il fine tuning degli iperparametri, e lo *shuffle*, in base al quale mischiare gli elementi del dataset (*True*) o ordinarli alfabeticamente (*False*).

In questa fase procediamo anche alla divisione tra training set e validation set, riservando il 30% per quest'ultimo grazie al parametro *validation\_split*.

Per quanto riguarda il test set, il caricamento viene effettuato allo stesso modo, con la differenza che non abbiamo bisogno di specificare i parametri *subset* e *validation\_split* in quanto non c'è divisione all'interno della cartella di test, e che il *batch\_size* è impostato al valore 1 per processare l'intero insieme di elementi in una sola iterazione.

```
1 train_ds = tf.keras.preprocessing.image_dataset_from_directory(  
2     directory=src_path_train,  
3     subset='training',  
4     labels='inferred',  
5     label_mode='categorical',  
6     class_names=['H', 'P'],  
7     image_size=(image_size, image_size),  
8     color_mode='grayscale',  
9     batch_size=batch_size,  
10    shuffle=False,  
11    validation_split=0.3,  
12 )
```

**Listato 2.2:** Caricamento del training set

## Verifiche sul dataset caricato

Per assicurarci che i campioni del dataset siano stati caricati correttamente, effettuiamo due diverse verifiche: una sulla shape dei Tensors delle immagini e delle etichette e un'altra visualizzando le immagini caricate. Il controllo è fatto sia sul training set che sul test set.

Il controllo sulle shapes è effettuato tramite un ciclo *for* sulle sezioni di training e test del dataset caricato: per ogni batch, stampiamo a video il risultato della chiamata al parametro *shape* sul dataset considerato.

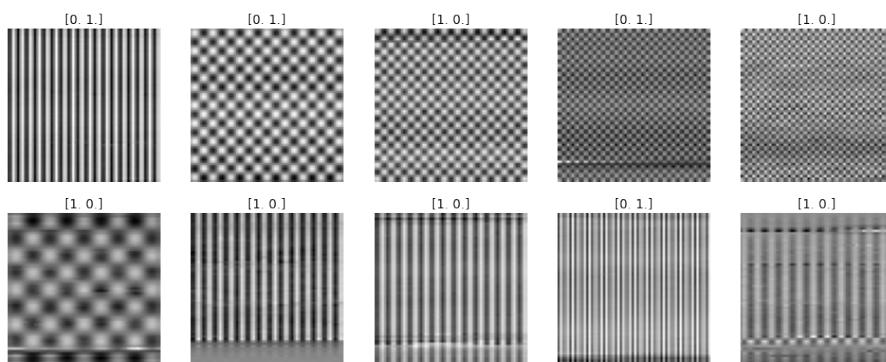
```
1 for image_batch, labels_batch in train_ds:
2     print(f'Shape delle immagini di training (batch): ', image_batch.shape)
3     print(f'Shape delle etichette di training (batch): ', labels_batch.shape)
4     break
```

**Listato 2.3:** Verifica delle shapes

Similmente, usiamo un ciclo *for* per iterare sugli elementi del dataset e visualizzarli grazie al modulo *pyplot* della libreria *matplotlib*. Nel dettaglio, prendiamo e mischiamo 100 campioni dal dataset tramite il metodo *shuffle* con parametro *buffer\_size* e visualizziamo così 5 elementi scelti casualmente, con la relativa etichetta rappresentante la classe come titolo ( $[1. 0.]$  = "Healthy",  $[0. 1.]$  = "Patient").

```
1 plt.figure(figsize=(15, 15))
2 for images, labels in train_ds.shuffle(buffer_size=5).take(1):
3     for i in range(5):
4         ax = plt.subplot(1, 5, i + 1)
5         plt.imshow(images[i], cmap=plt.get_cmap('gray'))
6         plt.title(np.array(labels[i]))
```

**Listato 2.4:** Visualizzazione delle immagini del training set



**Figura 2.1:** Esempio di campioni del dataset

### 2.1.3 Normalizzazione dei dati

Nel Machine Learning lo scaling dei dati, nel nostro caso specifico la normalizzazione, permette di minimizzare la variazione delle caratteristiche di un dataset. È una delle operazioni di preprocessing più utili in quanto migliora la qualità dei risultati finali. A parità di altre condizioni, infatti, lo scaling riduce il tempo in cui l'algoritmo di apprendimento converge al risultato finale, migliorando così l'efficacia del modello statistico. Inoltre, come vedremo nella sezione dedicata alla costruzione dei modelli, l'uso di funzioni di attivazioni non-lineari richiede la trasformazione dei dati originali per renderli compatibili con il nostro obiettivo.

Un altro motivo per normalizzare i dati in input è legato al problema dei gradienti: il rescaling dell'input entro un range di valori piccolo permette di avere valori piccoli anche per i pesi che vengono aggiornati durante il training, diminuendo le possibilità che l'output delle unità della rete si avvicinino alle regioni di saturazione delle funzioni di attivazione. Inoltre, ci permette di impostare il range iniziale di variabilità dei pesi in un intervallo molto limitato, nel nostro caso [0,1].

La normalizzazione avviene grazie all'utilizzo del layer *Rescaling* di Keras che scala i valori di input ad un nuovo range. Le immagini che compongono il nostro dataset sono costituite da valori nel range [0, 255] e verranno scalati al range [0, 1] in modo da poter utilizzare la funzione di attivazione *sigmoid* nel nostro modello.

```
1 normalization_layer = tf.keras.layers.Rescaling(scale=1./255)
2 normalized_train_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
```

**Listato 2.5:** Normalizzazione del training set

Il processo viene ripetuto anche per validation e test set ed il *rescaling* è applicato sia durante la fase di training che durante la fase di *inference*, cioè il processo di utilizzo del modello addestrato per fare previsioni. Nel nostro caso, trattandosi di una CGAN, questo processo corrisponde alla generazione di immagini sintetiche da parte del modello generatore addestrato.

## 2.2 Selezione ed adattamento dei modelli

Grazie all'esistenza di API per il Deep Learning come Keras [35], risulta abbastanza immediato utilizzare ed addestrare diversi modelli di Machine Learning su un determinato dataset. La sfida principale, pertanto, è rappresentata dalla scelta del modello da utilizzare e il suo conseguente adattamento dalla gamma di modelli

disponibili. Ingenuamente, si potrebbe pensare che sia sufficiente guardare la performance del modello, ma è necessario porre attenzione anche verso altri aspetti, come per esempio il tempo necessario all’addestramento o la facilità con la quale è possibile introdurlo a terzi.

La selezione dei modelli è il processo di scelta del modello di Machine Learning finale all’interno di una collezione di modelli candidati all’addestramento per un determinato dataset. Il processo può essere applicato sia a diversi tipi di modello (regressione, SVM, KNN, etc.) sia a modelli dello stesso tipo configurati con iperparametri di modello differenti. Nel caso in esame, per esempio, l’obiettivo è quello di sviluppare un modello capace di generare immagini sintetiche condizionate a partire da un dataset di immagini composto da pochi elementi. Non sappiamo in anticipo quale sia il modello con la migliore performance, dato che è impossibile saperlo. Pertanto, durante questo lavoro sono stati addestrati e valutati una grande varietà di modelli, al fine di selezionare quello più adatto al nostro problema specifico. Nel capitolo 3 verranno riportati i risultati di alcuni esperimenti provenienti dal processo di selezione.

Innanzitutto, è buona pratica dimenticare il concetto di modello "migliore". Tutti i modelli presentano errori di previsione, dato il rumore statistico nei dati, l’incompletezza dei campioni e le limitazioni legate ai diversi tipi di modello. Quindi, la nozione di modello perfetto o migliore non è utile, ma è utile cercare un modello che sia *good enough*, o sufficientemente valido. Anche questo concetto può avere diverse interpretazioni; un modello *good enough* può essere un modello tale da soddisfare le richieste di terzi, o che sia abbastanza valido in relazione al tempo e alle risorse disponibili, che sia valido rispetto ad altri modelli testati o allo stato dell’arte. Inoltre, come abbiamo visto nella sezione 2.1.3, alcuni modelli richiedono una preparazione dei dati particolare affinché possano esporre al meglio la struttura del problema all’algoritmo di apprendimento.

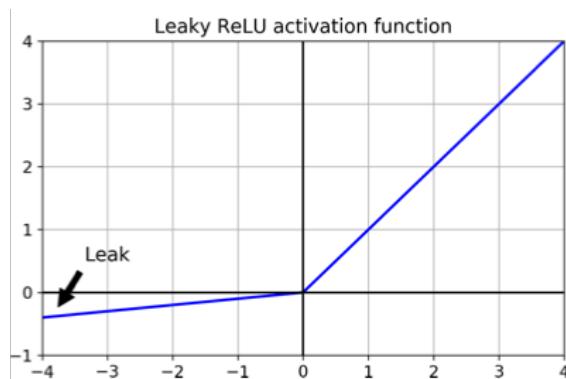
L’approccio migliore alla selezione dei modelli richiede una disponibilità di dati sufficiente, che potrebbe anche tendere ad infinito in base alla complessità del problema in esame. In questa situazione ideale, dividiamo il dataset in training, validation e test set, come visto nel paragrafo 2.1.2, addestriamo i modelli candidati sul training set, li valutiamo sul validation set e riportiamo le performance del modello finale sul test set. Dopo un lungo processo di selezione, si è deciso di utilizzare l’implementazione di una Deep Convolutional GAN (DCGAN) [36], poi adattata al nostro dataset, per entrambi i modelli: generatore e discriminatore.

### 2.2.1 Generatore

Come visto in precedenza, il generatore è responsabile della generazione di campioni sintetici che siano idealmente indistinguibili dai campioni reali del dataset. Nel nostro caso, i campioni corrispondono ad immagini con risoluzione 64x64 in scala di grigi e il dataset contiene anche informazioni aggiuntive, come le etichette di classe, che sfruttiamo per implementare la parte condizionale della CGAN e per generare in maniera mirata immagini di un determinato tipo.

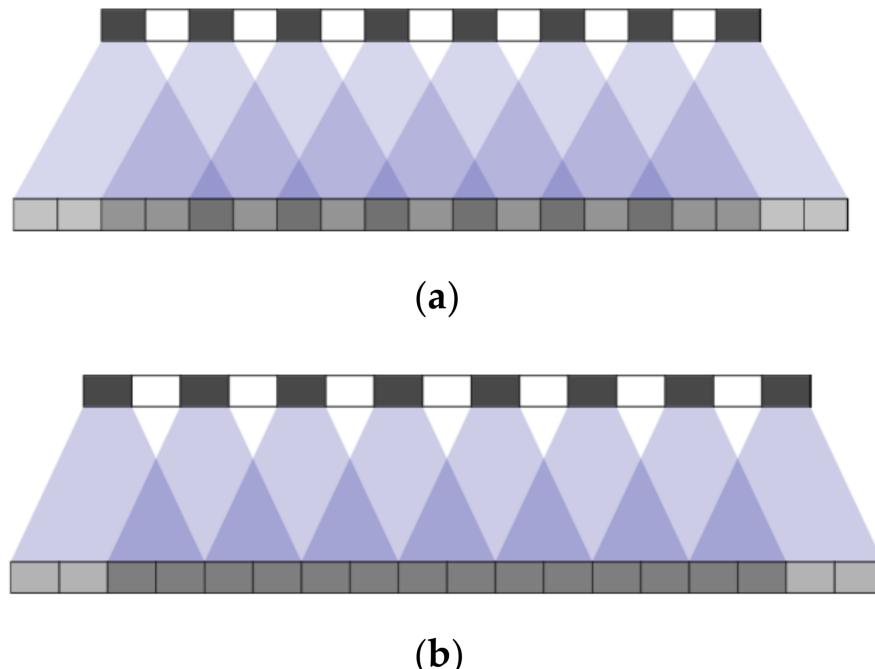
Il nostro generatore prende in input un punto dello spazio latente e l'etichetta relativa alla classe e restituisce come output una singola immagine 64x64 in scala di grigi. Ciò si ottiene usando un layer *fully connected* che interpreta il punto nello spazio e fornisce un numero sufficiente di attivazioni su cui possiamo eseguire un'operazione di *reshape* in diverse copie, nel nostro caso (128 + 2), di una versione a bassa risoluzione dell'immagine di output, per esempio 8x8 pixel. Sull'immagine vengono eseguite 3 operazioni di *upsampling*, triplicandone le dimensioni tramite i layer di trasposizione convoluzionale (o deconvoluzionali) e portandole da 8x8 a 64x64. Il modello fa uso di *best practices* relativi alle GAN come:

- Utilizzo di un layer di attivazione LeakyReLU - il Leaky Rectified Linear Unit è un tipo di funzione di attivazione basata su ReLU, ma che presenta una piccola pendenza per i valori negativi piuttosto che assumere il valore 0. È largamente utilizzato perché impostando l'attivazione a 0 per valori negativi può causare il problema della *dying ReLU* che porta all'*overfitting*.



**Figura 2.2:** Grafico della funzione di attivazione LeakyReLU [37]

- Dimensioni del kernel (4x4) di un fattore delle dimensioni dello *stride* (2x2)  
 - pratica utilizzata per evitare l'effetto *checkerboard*. Sfortunatamente, infatti, la deconvoluzione può avere un *overlap* spaiato quando le dimensioni del kernel (la finestra di output) non sono divisibili per le dimensioni dello stride (lo spostamento della finestra durante la deconvoluzione). Nonostante la rete possa, teoricamente, imparare assegnare con attenzione determinati pesi per evitare quest'effetto, in pratica è molto difficile evitarlo completamente. In figura 2.3 si evidenziano i due diversi casi di trasposizione convoluzionale: si nota come nel primo caso i pixel più scuri vengano "visitati" in diversi passi della finestra, creando l'effetto indesiderato [38].



**Figura 2.3:** Deconvoluzione - (a) stride=2, kernel=5; (b) stride=2, kernel=4 [39]

Riportiamo in figura 2.4 il riepilogo della rete grazie al metodo *Model.summary()*.

Model: "generator"		
Layer (type)	Output Shape	Param #
dense1 (Dense)	(None, 8320)	1089920
reshape1 (Reshape)	(None, 8, 8, 130)	0
t_conv1 (Conv2DTranspose)	(None, 16, 16, 128)	266368
relu1 (LeakyReLU)	(None, 16, 16, 128)	0
t_conv2 (Conv2DTranspose)	(None, 32, 32, 256)	524544
relu2 (LeakyReLU)	(None, 32, 32, 256)	0
t_conv3 (Conv2DTranspose)	(None, 64, 64, 512)	2097664
relu3 (LeakyReLU)	(None, 64, 64, 512)	0
output (Conv2D)	(None, 64, 64, 1)	12801

Total params: 3,991,297
Trainable params: 3,991,297
Non-trainable params: 0

**Figura 2.4:** Riepilogo del modello generatore

L'output del generatore è quindi una singola immagine sintetica che verrà sottoposta al "giudizio" del discriminatore.

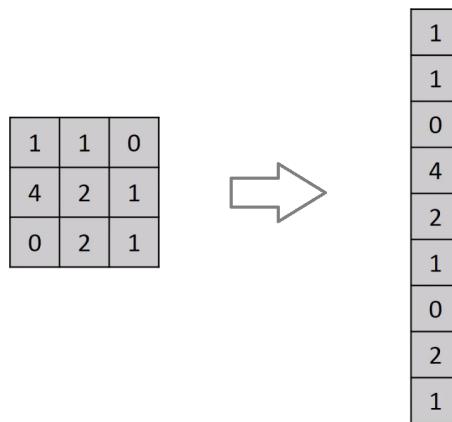
## 2.2.2 Discriminatore

Il discriminatore, come visto, è responsabile della classificazione dei campioni tra reali e sintetici. Nello specifico, cerca di distinguere tra le immagini reali provenienti dal training set utilizzato e le immagini sintetiche provenienti dal generatore, restituendo un feedback che permette al generatore di migliorare le immagini generate rendendole sempre più vicine alla distribuzione del dataset di partenza.

Il nostro discriminatore prende in input un’immagine 64x64 in scala di grigi e l’etichetta relativa alla classe di appartenenza del campione e restituisce come output una previsione binaria sulla natura dello stesso, cioè reale o sintetico. L’aggiunta dell’etichetta relativa alla classe del campione ha l’effetto di rendere l’immagine di input condizionale all’etichetta fornita.

Il modello è rappresentato da una rete CNN in cui avviene il processo inverso rispetto al generatore: i layer di attivazione, infatti, sono in questo caso preceduti da un layer convoluzionale, responsabile del *downsampling*. Dopo 3 layer convoluzionali, l’immagine ricevuta in input di dimensioni 64x64, grazie allo *stride* di dimensioni 2x2, diminuisce di un fattore 3 e raggiunge le dimensioni di 8x8 pixel. Anche per il discriminatore, quindi, abbiamo utilizzato le stesse *best practices* usate per il modello del generatore appena citate. In aggiunta, vengono utilizzati due nuovi layers, subito prima del layer di output:

- Layer *Flatten* - "appiattisce" l’input, rimuovendo tutte le dimensioni eccetto una. Ciò ha l’effetto di dare una nuova forma al tensore, affinché sia uguale al numero di elementi contenuti al suo interno. A tutti gli effetti coincide con la creazione di un array di elementi di una sola dimensione, senza includere la dimensione del batch: nel nostro caso infatti, si passa da una *shape* di (None, 8, 8, 128) a una *shape* di (None, 8192), dove 8192 è il risultato di 8x8x128. In questo modo riusciamo a connettere ogni elemento ai successivi elementi del layer *fully connected* evitando ambiguità e in modo efficace.



**Figura 2.5:** Esempio di layer *Flatten* [40]

- Layer *Dropout* - si occupa di impostare le unità di input a 0 in modo casuale ad ogni step durante il training, aiutando a prevenire l'*overfitting*. Da notare che il layer di *Dropout* si applica solo quando il parametro *training* è impostato su *True*, in modo che nessun valore venga rimosso durante l'*inference*. Utilizzando il metodo *model.fit* per addestrare la rete, infatti, il parametro viene abilitato automaticamente. Il valore di *rate* rappresenta la probabilità che i valori vengono rimossi, nel nostro caso del 20%

La figura 2.6 riporta l'output del metodo *Model.summary()*.

Model: "discriminator"		
Layer (type)	Output Shape	Param #
conv1 (Conv2D)	(None, 32, 32, 64)	3136
relu1 (LeakyReLU)	(None, 32, 32, 64)	0
conv2 (Conv2D)	(None, 16, 16, 128)	131200
relu2 (LeakyReLU)	(None, 16, 16, 128)	0
conv3 (Conv2D)	(None, 8, 8, 128)	262272
relu3 (LeakyReLU)	(None, 8, 8, 128)	0
flatten1 (Flatten)	(None, 8192)	0
dropout1 (Dropout)	(None, 8192)	0
output (Dense)	(None, 1)	8193

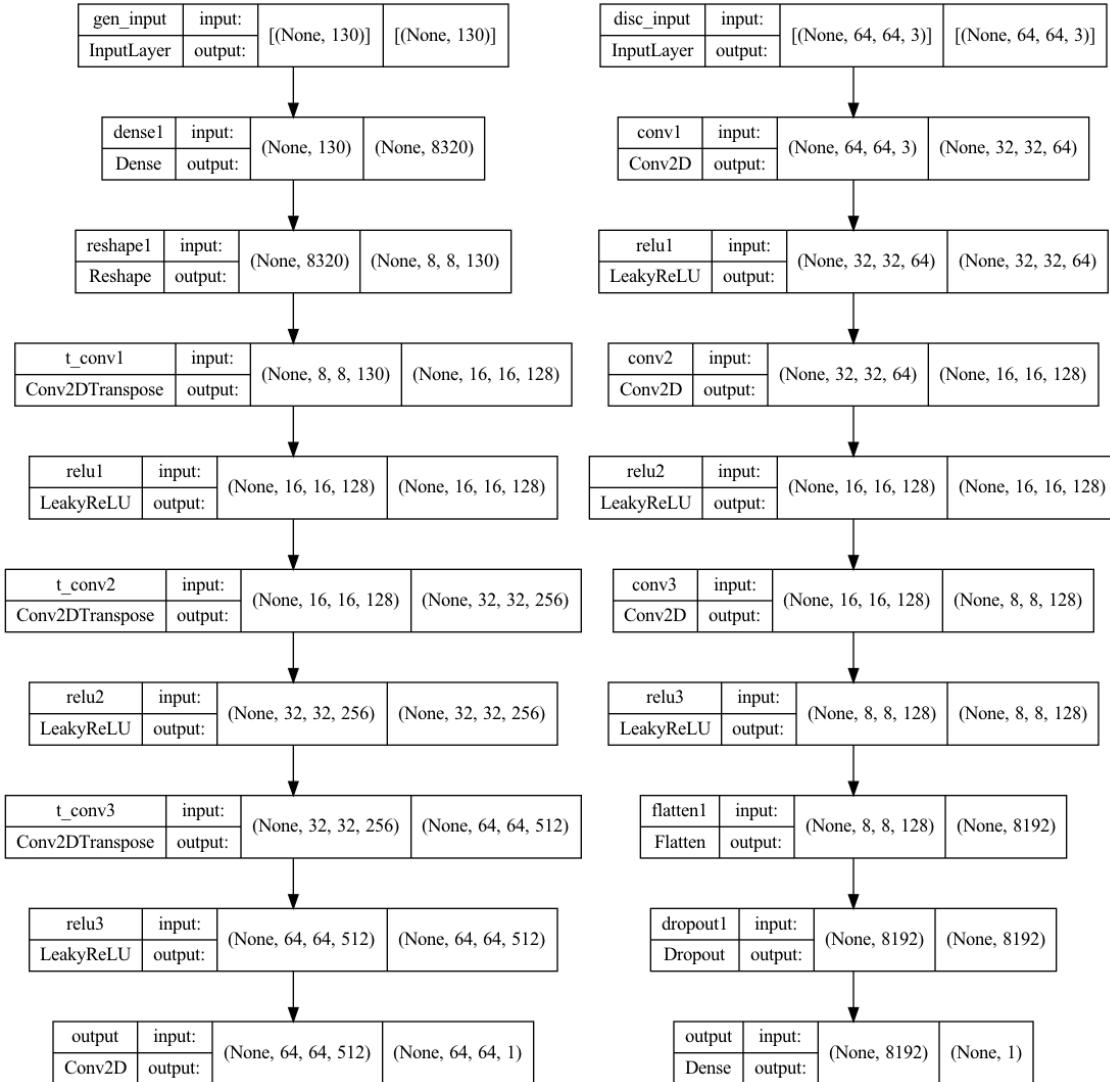
Total params:	404,801
Trainable params:	404,801
Non-trainable params:	0

**Figura 2.6:** Riepilogo del modello discriminatore

L'output del discriminatore è quindi una previsione binaria rispetto alla natura dell'immagine condizionale ad esso sottoposta, che sarà quindi reale o sintetica. Infatti, come visto nel paragrafo 2.1.3, l'output del discriminatore sarà un valore

compreso nell'intervallo [0,1] che rappresenta quindi la probabilità che l'immagine in esame sia reale o sintetica; per ottenere questo risultato facciamo uso della funzione di attivazione *sigmoid* nell'ultimo layer del modello.

Come riepilogo, riportiamo in figura 2.7 l'output del metodo *plot\_model()* su entrambi i modelli, che mostra i grafi delle reti con il tipo di ogni layer e la *shape* di input e output.



**Figura 2.7:** Grafo dei modelli generatore e discriminatore

### 2.2.3 Definizione del modello combinato

Dopo aver definito i modelli di generatore e discriminatore, è necessario costruire un modello che combini le due reti. Come anticipato, per questa fase abbiamo utilizzato l’implementazione presente sul sito ufficiale dell’API Keras [28] *as is*, pertanto non entreremo nei dettagli di questa sezione ma riportiamo per completezza i concetti fondamentali per il funzionamento della CGAN.

In primo luogo, le etichette in formato *one-hot-vector* (ottenute grazie al parametro *categorical* introdotto nella sezione 2.1.2) vengono concatenate alle immagini in input al discriminatore. Similmente, altre etichette vengono concatenate ai punti dello spazio latente prelevati dal generatore: quest’ultimo decodificherà il rumore, guidato dalle etichette, per generare le immagini sintetiche. Le due reti vengono poi addestrate grazie all’ API *GradientTape()* di Tensorflow, responsabile del calcolo dei gradienti durante il *forward pass* e il *backward pass*, e i pesi vengono aggiornati dopo ogni *train\_step* grazie alla *backpropagation* tramite le funzioni di costo dei due modelli, che sono le uniche funzioni tracciate nello script di riferimento: infatti, come accennato, l’implementazione della CGAN avviene effettuando un *subclassing* nella definizione della classe *ConditionalGAN* in questione e un override della funzione *train\_step* della classe *Model*, modificando a tutti gli effetti ciò che avviene nel metodo *Model.fit()* usato per avviare l’addestramento delle reti, rendendo possibile solo il tracciamento della *training loss* (funzione di costo di training) di generatore e discriminatore.

### Modifica del modello CGAN

Durante le fasi di selezione dei modelli già vista nel paragrafo 2.2 e di *fine-tuning* degli iperparametri presentata successivamente nel 2.3, si è reso necessario aggiungere altre funzioni di costo e metriche da tracciare per migliorare l’addestramento della rete. Per questo motivo abbiamo aggiunto allo script di riferimento, in momenti successivi, il tracciamento di funzioni di costo classiche nel campo del training delle reti neurali, come *validation loss* e *test loss* per generatore e discriminatore e di metriche classiche come *accuracy* e *mean squared error (MSE)* su tutte e tre le parti del dataset, cioè *training*, *validation* e *test set*.

Solitamente, Tensorflow e Keras permettono di tracciare funzioni di costo e metriche semplicemente specificandole nel metodo *compile()*. Quest’ultimo, infatti, può ricevere gli argomenti *loss* e *metrics*, in cui possiamo indicare rispettivamente

i nomi delle funzioni di costo da tracciare e le classi corrispondenti alle metriche scelte. I valori saranno poi mostrati a schermo durante l'esecuzione del metodo *fit()* e registrate nell'oggetto *History* restituito da *fit()*. Inoltre, sono restituite anche dal metodo *model.evaluate()* utilizzato nel paragrafo 2.5.1.

Nel nostro caso, essendo presente un *subclassing* (2.2.3) e un conseguente *override* del solo training step, non è stato possibile sfruttare questa possibilità; si è reso necessario, infatti, modificare lo script in due modi:

- Aggiunta delle metriche al training step - per tenere traccia di *accuracy* e *mse* abbiamo utilizzato il metodo *update\_state()*, responsabile dell'accumulo delle statistiche per l'elaborazione delle metriche, su etichette e previsioni del discriminatore.
- Override del *test step* - più articolata la soluzione per aggiungere il tracciamento di funzioni di costo e metriche su *validation* e *test set*: per portare a termine questo task, infatti, abbiamo dovuto eseguire un *override* sul test step, aggiungendo anch'esso alla definizione del modello combinato *ConditionalGAN*. In particolare, abbiamo ripetuto le operazioni effettuate nel training step, evitando di utilizzare la *backpropagation* e di eseguire le operazioni in *inference mode*, impostando il parametro *training* del metodo *update\_state* su *True*.

Sia per il training step che per il test step, abbiamo concluso il task restituendo tutte le metriche aggiunte tramite *return*, facendo così in modo che venissero visualizzate a schermo durante il training e registrate nell'oggetto *History*.

## 2.3 Ottimizzazione degli iperparametri

L'ottimizzazione o *tuning* degli iperparametri è il processo di determinazione della giusta combinazione di iperparametri che massimizzi la performance del modello addestrato su uno specifico dataset. Il frutto di questo processo è il set di valori più adatti affinché il modello in esame ottenga risultati ottimali; si evince, quindi, che il *tuning* è di fondamentale importanza ai fini dell'addestramento della rete.

Dopo un lungo processo di ottimizzazione, si è verificato che i valori ottimali al fine di ottenere un addestramento relativamente stabile e convergente della nostra CGAN sono i seguenti:

**Tabella 2.1:** Valori finali degli iperparametri

Divisione Training / Test sets	50% / 50%
Divisione Training / Validation sets	70% / 30%
Numero di seeds per ogni task	5
Batch size	8
Algoritmo di ottimizzazione	Adam
Learning rate (generatore e discriminatore)	0.00002
Decay rate (generatore e discriminatore)	0.9
Funzione di costo	Binary Cross Entropy
Epochs	500

I risultati relativi all’addestramento con gli iperparametri ottimali, oltre ad un esempio di training effettuato nelle fasi iniziali del processo di ottimizzazione, saranno presentati nel capitolo 3, dedicato all’esposizione dei risultati ottenuti.

Rispetto all’implementazione scelta come riferimento [28], anche il processo di *tuning* ha richiesto diverse modifiche. In primo luogo, la divisione dei campioni tra training, test e validation set e quella in 5 seeds è stata dettata dalla natura del dataset, fornito già con queste specifiche per limitare la casualità degli esperimenti.

La scelta del *batch size* dipende largamente dalla memoria della GPU sulla quale vengono effettuati gli esperimenti: nel nostro caso il valore 8 è un buon compromesso tra velocità dell’addestramento e il rischio di saturazione della memoria con conseguente crash, così come la scelta di limitarci a 500 epochs di training che permette di avere un buon bilanciamento tra performance e impiego di risorse.

La funzione di costo è rimasta invariata in quanto adatta al nostro task di classificazione binaria dei campioni, così come la scelta della versione Adam dell’algoritmo di ottimizzazione SGD, *best practice* per l’addestramento delle GAN in generale.

Gli iperparametri sui quali sono stati necessari più studi sono i *learning rates* dell’ottimizzatore Adam per entrambi i modelli. Come vedremo nel capitolo 3, è stato difficile ottenere un training stabile e convergente per diversi esperimenti, in quanto sia i valori di default (0.001) che quelli utilizzati nello script di riferimento (0.0003) risultavano troppo alti, causando un aggiornamento dei pesi del generatore troppo veloce su valori errati e portando velocemente ad una divergenza tra le funzioni di costo dei modelli.

## 2.4 Tracciamento dei progressi e dei risultati

Un aspetto fondamentale del Machine Learning e nello specifico dell’addestramento di Reti Neurali è la necessità di tener traccia degli esperimenti effettuati, soprattutto durante il lungo processo di *trial and error* e il *fine-tuning* dei modelli.

Durante il corso di questo lavoro sono stati utilizzati diversi metodi, anche contemporaneamente, per far sì che gli errori non si ripetessero in esperimenti successivi e per gestire al meglio i risultati ottenuti.

### 2.4.1 Callbacks API

Un *callback* [35] è un oggetto capace di eseguire azioni a vari stadi dell’addestramento (p.e. all’inizio o alla fine di un’epoch, prima o dopo un singolo *batch*).

Durante l’attività abbiamo ritenuto opportuno usufruire sia di *callbacks* di default di Keras, sia di *callbacks* personalizzati. Tra quelli già disponibili abbiamo utilizzato *CSVLogger()*, che ci permette di costruire un file .csv con i valori dei costi e delle metriche durante l’addestramento, e *ModelCheckpoint()* grazie al quale possiamo salvare i pesi del modello con una certa frequenza, p.e. ogni 10 epochs. Da notare che anche in questo caso il *subclassing* (2.2.3) causa delle limitazioni e non ci permette di salvare l’intero modello, con la conseguenza che una volta terminato il training è stato necessario caricare i pesi e ricompilare il modello per poterlo salvare ed utilizzare in altri ambiti. Il sistema di checkpoints è molto utile per le GAN in quanto spesso le prestazioni sono altalenanti e il training può migliorare prima di peggiorare e viceversa.

Oltre ai *callbacks* di default appena citati, abbiamo utilizzato un *callback* personalizzato grazie ad un override del metodo *on\_epoch\_end()*. Modificando ciò che avviene in questo metodo, nelle ultime fasi di questo lavoro abbiamo potuto verificare la qualità delle immagini sintetiche generate durante il training: generando un set di immagini ogni 10 epochs, in concomitanza con il checkpoint del modello (come si evince dall’esempio in figura 2.8), riusciamo a valutare la qualità delle stesse e ad avere la possibilità di caricare successivamente un modello al momento in cui generava le immagini migliori, oppure semplicemente fermarlo quando le immagini inizino ad avere una qualità peggiore, per risparmiare tempo e risorse; ciò è fondamentale per le GAN in quanto, pur essendo gli studi ancora in corso, spesso l’ispezione visiva dei campioni generati rappresenta uno tra i metodi più intuitivi per valutare le prestazioni del modello. [41] [42]

Il listato 2.6 mostra la porzione di codice relativa ai *callbacks* personalizzati.

```
1 class GANMonitor(tf.keras.callbacks.Callback):
2     def __init__(self, n_img=5, latent_dim=generator_in_channels):
3         self.n_img = n_img
4         self.latent_dim = latent_dim
5     # override del metodo chiamato alla fine di ogni epoch
6     def on_epoch_end(self, epoch, logs=None):
7         random_latent_vectors = tf.random.normal((self.n_img, self.latent_dim))
8         generated_images = self.model.generator(random_latent_vectors)
9         generated_images *= 255
10        generated_images.numpy()
11        for i in range(self.n_img):
12            img = tf.keras.preprocessing.image.array_to_img(generated_images[i])
13            if (epoch+1) % 10 == 0:
14                img.save("synth_{}_{:03d}.png".format(i, epoch+1))
15                print("Immagine generata: synth_{}_{:03d}.png".format(i, epoch+1))
```

**Listato 2.6:** Custom callback per la generazione di immagini durante il training

### Tracciamento con Weights and Biases

L'ultimo *callback* utilizzato è *WandbCallback()*, proveniente dal modulo *wandb* [31] menzionato nella sezione 2.1.1 di importazione delle librerie. Il modulo ci permette di tenere traccia comodamente dei risultati degli esperimenti, grazie alle funzioni *wandb.init*, che inizializza un nuovo test all'inizio dello script, *wandb.config* che traccia iperparametri e metadati, e *wandb.log* che salva i valori delle metriche registrate nel training *loop*.

## 2.5 Addestramento della CGAN

La fase di ottimizzazione degli iperparametri esplorata nel paragrafo 2.3 è interval-lata dal processo di addestramento della rete, affinché si possa verificare che le scelte effettuate portino ad un training stabile e convergente della GAN. Quest'ultimo si avvia al termine, nel nostro caso, di tre fasi:

- Dichiarazione del modello combinato *ConditionalGAN* definito nel 2.2.3.
- Configurazione del modello combinato appena definito, grazie all'utilizzo del metodo *Model.compile()*.
- Chiamata del metodo *Model.fit()*, responsabile del training, sul modello appena compilato.

Nel listato 2.7 riportiamo il codice relativo alle tre fasi; da notare anche la presenza, nella chiamata del metodo *fit()*, del parametro *callbacks* che abbiamo esposto nel paragrafo 2.4.1.

```

1 cond_gan = ConditionalGAN(discriminator=d, generator=g, latent_dim=ld)
2 cond_gan.compile(
3     d_opt=tf.keras.optimizers.Adam(learning_rate=0.00002, beta_1=0.9),
4     g_opt=tf.keras.optimizers.Adam(learning_rate=0.00002, beta_1=0.9),
5     loss_fn=tf.keras.losses.BinaryCrossentropy(),
6     train_mse_metric=tf.keras.metrics.MeanSquaredError(name='mse'),
7     train_acc_metric=tf.keras.metrics.BinaryAccuracy(name='accuracy')
8 )
9 cond_gan.fit(
10    normalized_train_ds,
11    epochs=n_epochs,
12    validation_data=(normalized_val_ds),
13    validation_steps=(validation_samples/batch_size),
14    callbacks=[GANMonitor(n_img=5), get_callbacks(log_dir=log_dir), WandbCallback()]
15 )

```

**Listato 2.7:** Configurazione del modello ed inizio dell’addestramento

La figura sottostante rappresenta lo screenshot della finestra di VSCode durante il progresso del training. Si evincono i valori delle funzioni di costo e delle metriche aggiunte grazie all’*override* del test step (2.2.3), oltre ad alcuni riferimenti ai *callbacks* dell’API Keras, come il salvataggio delle immagini sintetiche e dei checkpoints dei modelli ogni 10 iterazioni (2.4.1).

```

Epoch 40/500
12/12 [=====] - ETA: 0s - g_loss: 0.7755 - d_loss: 0.7483 - mse: 0.2547 -
accuracy: 0.6250
Immagine generata: synth_0_epo40.png
Immagine generata: synth_1_epo40.png
Immagine generata: synth_2_epo40.png
Immagine generata: synth_3_epo40.png
Immagine generata: synth_4_epo40.png

Epoch 40: saving model to /Users/dylzen/Documents/Neural Networks/keras_cgan_-accXaccY/logs/
model_checkpoints/20220601-202005/cp-epo040-g_loss0.78-d_loss0.75-acc0.62-mse0.25
12/12 [=====] - 5s 463ms/step - g_loss: 0.7755 - d_loss: 0.7483 - mse: 0.2547 -
accuracy: 0.6250 - val_g_loss: 0.8454 - val_d_loss: 0.5625 - val_mse: 0.1884 - val_accuracy: 0.8542
Epoch 41/500
10/12 [=====>.....] - ETA: 0s - g_loss: 1.1601 - d_loss: 0.4628 - mse: 0.1283 -
accuracy: 0.8875

```

**Figura 2.8:** Progresso dell’addestramento della CGAN

### 2.5.1 Test del modello addestrato

In generale, dopo aver addestrato il modello è buona norma effettuare un test delle sue prestazioni su una porzione del dataset che il modello non ha mai elaborato: il test set. Nel caso specifico delle GAN e in riferimento al paragrafo 1.3.1, possiamo dire certamente che l'apprendimento è Non Supervisionato nel caso del generatore, ma risulta Supervisionato nel caso della Rete Avversaria. Per questo motivo, possiamo supporre che sia utile testare l'abilità del discriminatore nel distinguere campioni reali e sintetici su una parte del dataset che non ha mai visto prima. In altre parole, ha senso dividere il dataset in training-validation-test nel momento in cui si vuole capire quanto il discriminatore sia abile a generalizzare il suo compito su dati completamente nuovi.

A questo scopo, possiamo utilizzare il metodo *Model.evaluate()* del modello appena addestrato sul test set normalizzato nel paragrafo 2.1.3, restituendo inoltre i valori dei costi e delle metriche in formato *dictionary* con ciascuna chiave corrispondente al nome della metrica.

```
1 cond_gan.evaluate(normalized_test_ds, return_dict=True)
```

**Listato 2.8:** Valutazione del modello sul test set

Nel capitolo successivo verranno presentati i risultati degli esperimenti; effettueremo una valutazione quantitativa espressa grazie all'aiuto della dashboard di *wandb* [31] e una valutazione qualitativa ottenuta tramite ispezione visiva delle immagini sintetiche generate durante sia durante il training, grazie al *callback* personalizzato esplorato nel paragrafo 2.4.1, sia dal modello finale addestrato grazie ad un semplice script.

# Capitolo 3

## Valutazione dei risultati ed ulteriori esperimenti

In questo capitolo sono presentati i risultati ottenuti in vari stadi del lavoro svolto. Innanzitutto, è stata fatta una valutazione quantitativa del modello finale con il supporto di tabelle e grafici generati da Wandb [31], la dashboard specializzata per il tracciamento di esperimenti di Machine Learning (ML) introdotta nel paragrafo 2.4.1; è stato mostrato come la valutazione quantitativa finale, nel caso delle GAN, possa differire dalla valutazione qualitativa effettuata sulle immagini sintetiche generate alla fine dell'addestramento.

A valle delle valutazioni fatte sui modelli e sugli iperparametri presentati nel capitolo 2, si è reso necessario effettuare ulteriori esperimenti: dal paragrafo 3.3 saranno mostrati i relativi risultati che porteranno alle considerazioni finali.

### 3.1 Ottimizzazione dei modelli

Come anticipato nella sezione 2.3 relativa all'ottimizzazione degli iperparametri dei modelli, la prima *milestone* di questo lavoro è stata quella di ottenere un addestramento relativamente stabile e convergente della CGAN. Il dataset, come descritto nel 2.1.2, è stato fornito già diviso in seeds composti da un numero di campioni molto simile tra loro; per questo motivo, per la fase di *fine tuning* si è scelto di addestrare i modelli su uno solo dei seeds per entrambi tasks, in particolare il seed 1 del task *spiral* e il seed 2 del task *meander*, limitando il consumo di risorse.

Riportiamo un esempio di addestramento in questa prima fase, in cui non erano stati ancora individuati valori degli iperparametri tali da rendere le funzioni di costo (*loss functions*) convergenti e l’addestramento stabile; per questo particolare training, infatti, i valori scelti per il *learning rate* degli algoritmi di ottimizzazione di entrambi i modelli sono riportati in tabella 3.1, insieme agli altri parametri rilevanti.

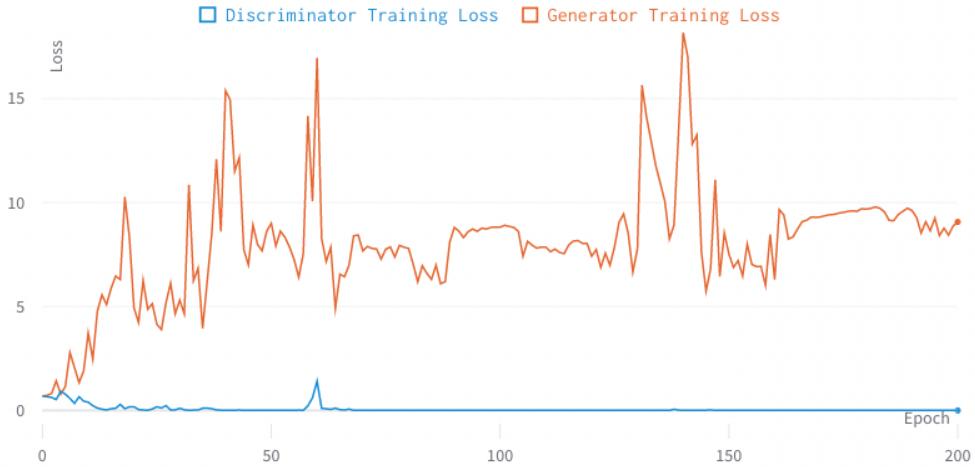
**Tabella 3.1:** Esempio di parametri scelti durante la fase di *tuning*

Batch size	8
Algoritmo di ottimizzazione	Adam
Learning rate (generatore e discriminatore)	0.0003
Decay rate (generatore e discriminatore)	0.9
Funzione di costo	Binary Cross Entropy
Epochs	500
Patience (Early Stopping)	200

Rispetto ai valori ottimizzati riportati nella tabella 2.1, il valore del *learning rates* è di un ordine di grandezza maggiore: ciò comporta un apprendimento più veloce da parte di entrambe le reti, discriminatore e generatore, durante il training. Infatti, l’algoritmo di ottimizzazione *Adam* utilizzato è un tipo di algoritmo *Stochastic Gradient Descent (SGD)*, responsabile dell’aggiornamento dei *weights* dei modelli grazie all’uso della *backpropagation*, cioè la propagazione all’indietro attraverso la rete, quindi dall’*output layer* all’*hidden layer*, dell’errore calcolato tra gli output attesi e gli output propagati in avanti dalla rete, o *forward propagated*. Una volta calcolati gli errori per ogni nodo della rete, essi vengono usati per aggiornare i *weights* facendo uso anche del *learning rate*, che rappresenta la velocità dell’addestramento.

In queste fasi iniziali, come evidenziato dalla tabella 3.1, è stato usato anche il meccanismo dell’*Early Stopping*, facente anch’esso parte delle *callbacks API*, grazie al quale il training viene arrestato dopo un numero specifico (*patience*) di *epochs* in cui il trend della funzione di costo non presenta miglioramenti.

L’effetto di un *learning rate* troppo elevato è evidenziato nel grafico 3.1 relativo a questo addestramento: i pesi per entrambi i modelli sono aggiornati troppo velocemente con valori errati portando ad una divergenza immediata delle funzioni di costo, con la *loss function* del discriminatore subito tendente a 0.



**Figura 3.1:** Funzioni di costo di  $D$  e  $G$  per il seed *meander02* prima dell'ottimizzazione

In termini pratici, l'andamento delle due funzioni di costo mostrate indica l'inabilità del generatore di produrre immagini sintetiche accettabili perché l'addestramento molto rapido del discriminatore non gli permette di ricevere un feedback adeguato per poter migliorare il suo output.

## 3.2 Valutazione del modello finale

### 3.2.1 Valutazione quantitativa

Dopo il lungo processo di *trial and error* e alla fine della fase di *fine tuning*, è stato possibile raggiungere una relativa convergenza e stabilità dell'addestramento sui seeds *spiral01* e *meander02*. Lo studio delle GAN è ancora in corso ma il consenso in letteratura [43] è che un training si possa considerare stabile se il costo del discriminatore si mantiene intorno al valore 0.5 e il costo del generatore intorno al valore 0.9. Oltre alle funzioni di costo, l'obiettivo è anche quello di raggiungere un buon valore dell'*accuracy* nella fase di training, che per le GAN dovrebbe rimanere intorno al 70%; una percentuale più bassa, intorno al 50%, denoterebbe uno scarso addestramento del discriminatore che porta a delle previsioni equivalenti al lancio di una monetina, mentre una percentuale che si avvicina al 100% denoterebbe uno scarso addestramento del generatore, portando alla generazione di campioni sintetici palesemente falsi ed individuati facilmente dal discriminatore.

## Fase di addestramento

Raggiunti gli obiettivi prefissati sui due seeds di riferimento, la rete CGAN è stata addestrata su ognuno dei 10 seeds che compongono i 2 tasks, utilizzando per gli iperparametri i valori presentati nella tabella 2.1. La divisione esatta per training, validation e test set è riportata nella tabella 3.2

**Tabella 3.2:** Suddivisione del dataset per ogni seed

Seed	Numero di campioni		
	Training set	Validation set	Test set
<b>meander01</b>	93	38	131
<b>meander02</b>	96	40	128
<b>meander03</b>	96	39	127
<b>meander04</b>	90	37	135
<b>meander05</b>	90	37	135
<b>spiral01</b>	93	38	131
<b>spiral02</b>	96	39	127
<b>spiral03</b>	96	39	127
<b>spiral04</b>	90	37	135
<b>spiral05</b>	90	37	135

Riportiamo in tabella 3.3 e 3.4 i valori ottenuti con l’addestramento dopo la fase di ottimizzazione su ogni seed all’epoch 1000.

**Tabella 3.3:** Costi e metriche registrate nella fase di training per il task *meander* all’epoch 1000

Seed	Training set				Validation set			
	D Loss	G Loss	Accu.	MSE	D Loss	G Loss	Accu.	MSE
<b>me01</b>	0.57	0.89	0.77	0.19	0.64	1.00	0.66	0.22
<b>me02</b>	0.56	0.89	0.66	0.18	0.66	1.03	0.64	0.23
<b>me03</b>	0.54	0.93	0.75	0.19	0.52	1.05	0.85	0.16
<b>me04</b>	0.56	0.82	0.75	0.19	0.64	0.85	0.70	0.22
<b>me05</b>	0.63	0.80	0.65	0.21	0.63	0.83	0.77	0.22
<b>Media</b>	0.57	0.87	0.72	0.19	0.62	0.95	0.72	0.21

**Tabella 3.4:** Costi e metriche registrate nella fase di training per il task *spiral* all’epoch 1000

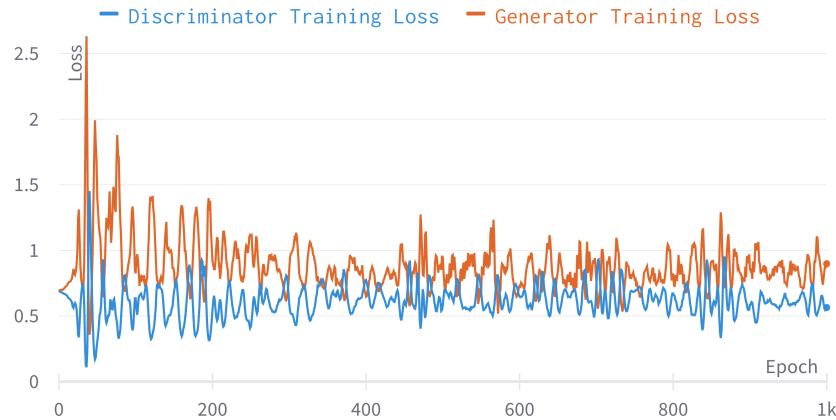
Seed	Training set				Validation set			
	D Loss	G Loss	Accu.	MSE	D Loss	G Loss	Accu.	MSE
sp01	0.65	0.99	0.62	0.21	0.77	0.91	0.62	0.28
sp02	0.56	0.90	0.72	0.20	0.77	1.30	0.56	0.27
sp03	0.58	0.89	0.71	0.20	0.58	0.97	0.70	0.19
sp04	0.63	0.88	0.70	0.21	0.71	0.90	0.64	0.25
sp05	0.57	0.85	0.75	0.19	0.60	0.95	0.75	0.20
Media	0.59	0.90	0.70	0.20	0.68	1.00	0.65	0.23

È importante evidenziare che, trattandosi di reti GAN, è sempre necessario osservare anche gli andamenti dei costi e delle metriche tramite grafici, mostrati successivamente; per natura dell’architettura (1.4.1), infatti, i valori registrati ad una determinata *epoch* possono differire di molto rispetto all’*epoch* precedente o successiva, soprattutto per quanto riguarda l’*accuracy*.

Fatta questa dovuta premessa, i valori ottenuti alla fine dell’*epoch* 1000 sono in linea con i risultati sperati:

- Le funzioni di costo sul training set di entrambi i modelli rientrano nel range di valori per i quali il training si può considerare stabile.
- L’*accuracy* sul training set risulta buona, con una media del 72% per il task *meander* e del 70% per il task *spiral*.
- Il valore del *MSE* è molto basso per tutti i seeds; il risultato è atteso in quanto, per le reti neurali in generale, ad un’*accuracy* alta deve corrispondere un *MSE* basso, denotando così una buona corrispondenza tra il dataset e le previsioni.
- La distanza delle funzioni di costo tra training e validation sets resta bassa, escludendo così la possibilità che si sia verificato un *overfitting*. Un esempio pratico è mostrato in figura 3.3
- Valori dell’*accuracy* sul validation set in linea con quelli registrati per il training set, e questo punto potrebbe indicare una buona performance del modello anche su un dataset mai elaborato prima.

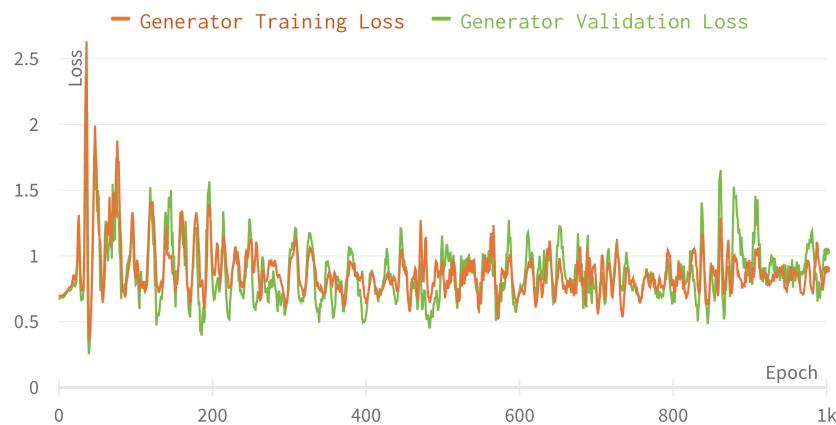
In figura 3.2 è mostrato come esempio l’andamento delle funzioni di costo di discriminatore e generatore sul training set per il seed *meander02*.



**Figura 3.2:** Funzioni di costo di  $D$  e  $G$  sul training set per il seed *meander02*

Dopo una fase di assestamento iniziale, entrambe le funzioni tendono a restare in un intorno dei valori presi come obiettivo. L’andamento altalenante e la varianza sono caratteristiche tipiche delle *loss functions* delle GAN in quanto le due reti sono in competizione e il miglioramento di una rete comporta inevitabilmente il peggioramento della rete avversaria.

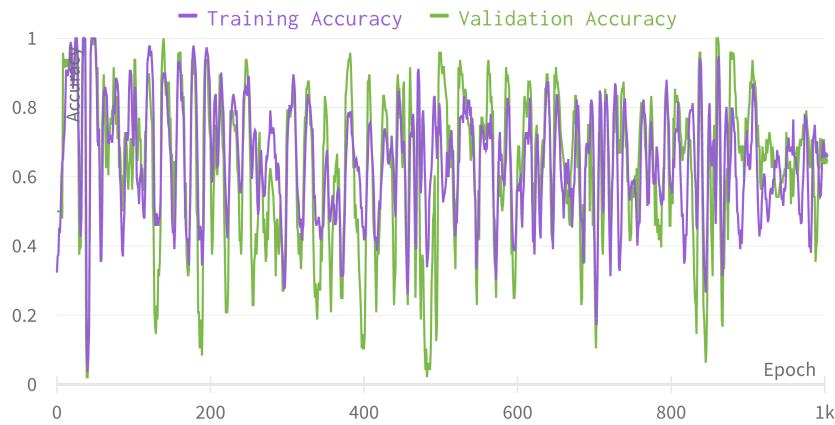
In figura 3.3 l’andamento di *training loss* e *validation loss* di  $G$ .



**Figura 3.3:** Funzioni di costo di  $G$  su training e validation set per il seed *meander02*

Come anticipato, i valori finali all’*epoch* 1000 ma soprattutto l’andamento delle funzioni di costo del generatore su training e validation set non denotano la presenza di *overfitting* dei modelli. L’argomento è ancora molto discusso in letteratura, in quanto molte pubblicazioni mostrano che le GAN in generale non sono soggette ad *overfitting*, in quanto il generatore  $G$  non può memorizzare i dati del training set perché non apprende direttamente dai campioni di training ma dal feedback del discriminatore  $D$  [44], mentre altre mostrano come l’*overfitting* sia possibile anche per le GAN [45].

La figura 3.4 mostra l’andamento dell’*accuracy* su training set e validation set: anche in questo caso il risultato è quello atteso in quanto tipico dell’addestramento di una GAN.



**Figura 3.4:** Accuracy di  $D$  su training e validation set per il seed *meander02*

All’inizio del training l’*accuracy* presenta una varianza molto alta, in quanto il generatore passa dal produrre immagini composte quasi esclusivamente da rumore, facilmente riconoscibili dal discriminatore, a produrre immagini totalmente casuali che il discriminatore non addestrato fatica a riconoscere. Intorno all’*epoch* 900 l’*accuracy* su entrambi i set tende a convergere al valore 0.7, con una varianza che diminuisce con il tempo; in generale, ciò significa che l’addestramento della GAN sta procedendo bene e che la sua qualità finale, rispecchiata nelle metriche sul validation set, sarà vicina a quella desiderata.

## Fase di test

Il metodo *Model.evaluate()* utilizzato come da paragrafo 2.5.1, ci permette di valutare le prestazioni della rete su una porzione di dataset sconosciuta, elaborata per la prima volta in questa fase.

La tabelle 3.5 e 3.6 mostrano i risultati ottenuti dalla CGAN sul test set per ogni seed.

**Tabella 3.5:** Costi e metriche registrate nella fase di test per il task *meander*

Seed	Test set			
	D Loss	G Loss	Accuracy	MSE
<b>me01</b>	0.65	0.84	0.68	0.22
<b>me02</b>	0.66	0.93	0.65	0.23
<b>me03</b>	0.56	0.94	0.77	0.19
<b>me04</b>	0.66	0.87	0.66	0.23
<b>me05</b>	0.68	0.77	0.64	0.24
<b>Media</b>	0.64	0.87	0.68	0.22

**Tabella 3.6:** Costi e metriche registrate nella fase di test per il task *spiral*

Seed	Test set			
	D Loss	G Loss	Accuracy	MSE
<b>sp01</b>	0.74	0.72	0.51	0.26
<b>sp02</b>	0.75	1.20	0.58	0.25
<b>sp03</b>	0.61	0.92	0.68	0.20
<b>sp04</b>	0.76	0.77	0.50	0.28
<b>sp05</b>	0.65	1.01	0.63	0.22
<b>Media</b>	0.70	0.92	0.58	0.24

Le prestazioni del modello sono buone, ma si evince una leggera differenza di performance tra i due tasks: il valore medio dell’*accuracy*, per esempio, è del 58% per il task *spiral* contro i 68% del task *meander*. Il valore relativamente basso registrato per il task *spiral* denota uno scarso addestramento del discriminatore, che di fronte ad immagini mai viste prima non riesce ad effettuare una previsione corretta, addirittura nel 50% dei casi per il seed 4.

## Considerazioni finali sull’addestramento

Il valore basso registrato sull’*accuracy* nella tabella 3.6 suggerisce di fare delle considerazioni che valgono soprattutto per la fase di training mostrata nel 3.2.1. Un’*accuracy* bassa del discriminatore, infatti, può essere ricondotta a due motivi, dai quali si evincono sia la difficoltà nell’addestrare una GAN che la poca affidabilità delle metriche classiche utilizzate per effettuare una valutazione quantitativa:

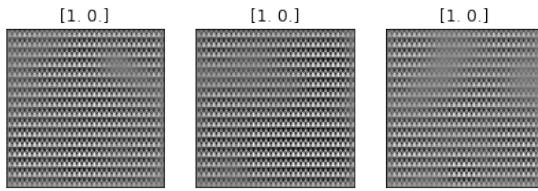
- Il generatore è addestrato in modo tale da produrre immagini sintetiche indistinguibili dai campioni del dataset. In questo caso, il discriminatore non è capace di riconoscere le immagini reali da quelle generate e la sua *accuracy* tende al 50% (vedi seed sp04). Questa rappresenta un addestramento ideale e grazie al processo di *data augmentation* diventa possibile aggiungere nuove immagini al dataset originale, raggiungendo l’obiettivo di questo lavoro. È importante notare che la progressione dell’addestramento rappresenterebbe la fonte di un problema per la convergenza della GAN: il feedback del discriminatore diventa meno significativo con il passare del tempo [46]. Infatti, se l’addestramento dovesse continuare oltre il punto in cui il discriminatore inizia a fornire un feedback completamente casuale, il generatore comincerebbe a lavorare su questo feedback senza significato e la sua qualità potrebbe collassare.
- Il generatore non riesce ad imparare la distribuzione dei campioni del dataset e di conseguenza produce immagini sintetiche di scarsa qualità. Generalmente, si vuole che la GAN produca un’alta varietà di output; se il generatore, nelle fasi iniziali del training in cui anche il discriminatore non è addestrato a sufficienza, produce un output plausibile secondo il discriminatore, potrebbe iniziare a produrre solo quel particolare output. In questo caso, la strategia migliore per il discriminatore è quella di imparare a rifiutare sempre quel l’output; ma se la versione successiva del discriminatore rimane bloccata in un minimo locale e non trova la strategia migliore, diventa facile per il generatore successivo trovare l’output più plausibile per il discriminatore corrente.

Ogni iterazione del generatore ottimizza eccessivamente per un particolare discriminatore, che non riesce mai ad uscire dalla trappola; il risultato è che il generatore continua a produrre un numero limitato di outputs e l’*accuracy* si abbassa. Questa forma di *failure* nelle GAN è chiamata *mode collapse*.

### 3.2.2 Valutazione qualitativa

Le considerazioni fatte nel 3.2.1 hanno reso necessaria una valutazione qualitativa dell’addestramento della CGAN, effettuata tramite ispezione visiva delle immagini generate. Questo metodo di valutazione delle GAN è molto diffuso [47], anche se presenta diverse limitazioni [48], soprattutto quando chi si occupa dell’ispezione non conosce bene il dominio di appartenenza dei dati.

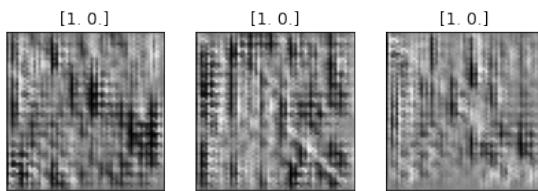
Utilizzando le *Callbacks API* descritte nel 2.4.1, abbiamo generato un set di immagini dopo ogni *epoch* del training. In figura è mostrato un esempio di immagini generate durante il training sul seed *meander02*, nello specifico all’*epoch* 40.



**Figura 3.5:** Esempio di immagini prodotte dal generatore all’*epoch* 40

Le immagini generate durante il training, come si evince facilmente, hanno una qualità scarsa; sono lontane dalla distribuzione del training set e soprattutto sono molto simili tra loro, presentando chiari segni di *mode collapse*, fenomeno descritto nel paragrafo 3.2.1.

In figura 3.6 è mostrato un set di immagini generate alla fine dell’addestramento, all’*epoch* 1000.



**Figura 3.6:** Esempio di immagini prodotte dal generatore alla fine del training

I risultati sono chiari: anche le immagini generate alla fine del training sono di bassissima qualità, ancora più lontane dai campioni del training set, e continuano ad evidenziare il *mode collapse* riscontrato già all’inizio dell’addestramento.

### 3.3 Data augmentation

Dopo aver constatato la scarsa qualità delle immagini generate, sono state fatte ricerche su quale potesse esserne la causa. Fin da subito, l'attenzione è stata posta alla radice degli esperimenti: il dataset.

Durante le ricerche è stato appurato che le GAN richiedono un elevato numero di campioni nel training set affinché il loro addestramento vada a buon fine; come visto in tabella 3.2, è evidente che il dataset convertito dal *NewHandPD* di Pereira et al. [7] a nostra disposizione non rispetta questa caratteristica.

Per dimostrare che il numero estremamente basso di campioni del dataset influenza negativamente sulla qualità delle immagini generati durante e alla fine dell'addestramento, è stata fatta un'operazione di *data augmentation* sul seed 2 del task *meander* prima del training; si tratta di una tecnica utilizzata per aumentare la diversità del training set applicando trasformazioni realistiche casuali, come per esempio la rotazione delle immagini [49]. Utilizzando i metodi di *tf.image* di TensorFlow, abbiamo applicato le trasformazioni ai soli campioni del training set, escludendo validation e test set.

In riferimento alla tabella 3.2 che indica le divisione del dataset, sono stati utilizzati i 96 campioni del training set di *meander02* per creare 15 nuovi dataset diversi grazie a 15 combinazioni di trasformazioni differenti.

Il listato 3.1 riporta un esempio di trasformazioni applicate al dataset originale.

```
1 def augment_15(image):
2     image = tf.image.rot90(image, k=3)
3     image = tf.image.flip_left_right(image)
4     image = tf.image.flip_up_down(image)
5     return image
```

**Listato 3.1:** Esempio di trasformazioni applicate al dataset per l'*augmentation*

Dopo aver definito le trasformazioni da applicare al dataset, esse vengono applicate al dataset tramite *Dataset.map* che crea a tutti gli effetti 15 ulteriori dataset composti da immagini aumentate.

Il listato 3.2 riporta un esempio di creazione di uno dei dataset aumentati, in cui viene anche utilizzato il parametro *num\_parallel\_calls* che configura il dataset per prestazioni migliori con l'uso di letture parallele e del *buffered prefetching*.

```
1 normalized_train_ds_15 = (
2     train_ds.map(augment_15, num_parallel_calls=AUTOTUNE)
3 )
```

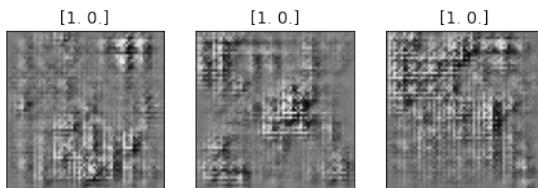
**Listato 3.2:** Creazione di un nuovo training set aumentato tramite trasformazioni

Una volta mappati i 15 datasets aumentati, essi vengono aggiunti al dataset originale di 96 campioni tramite il metodo `concatenate()` di `tf.data.Dataset`; in questo modo abbiamo ottenuto esattamente 1536 diverse immagini su cui effettuare il training.

### 3.3.1 Risultati del training con iperparametri di default

L’addestramento sul nuovo seed 2 del task *meander* è stato fatto limitandolo a 100 *epochs* a causa dell’aumento considerevole del training set. Tutti gli altri iperparametri sono rimasti invariati rispetto a quelli utilizzati prima dell’*augmentation* in tabella 2.1.

Come si evince dalla figura 3.7, le immagini generate presentano gli stessi problemi di *mode collapse* e di scarsa qualità evidenziati nel 3.2.2.



**Figura 3.7:** Esempio di immagini prodotte da  $G$  addestrato sul seed *meander02* aumentato

### 3.3.2 Ulteriore tuning degli iperparametri

Alla luce dei risultati ottenuti, è stato fatto un ultimo tentativo per cercare di migliorare l’output del generatore, ossia un’ulteriore fase di *fine tuning*, molto più breve rispetto alla fase osservata nel 2.3; in particolare, considerato l’aumento considerevole del numero di campioni di training, è stato deciso di modificare i parametri dell’algoritmo di ottimizzazione *Adam*, aumentando il *learning rate* per discriminatore e generatore e riducendo contemporaneamente il *decay rate* di entrambi. Le due modifiche apportate agli ottimizzatori hanno l’effetto di velocizzare l’addestramento dei modelli e di mantenerlo alto per un tempo più lungo.

Gli iperparametri finali sono riportati in tabella 3.7.

Un’ultima valutazione qualitativa delle immagini generate dal modello appena addestrato è presentata nel paragrafo successivo, dedicato alla generazione condizionale delle stesse.

**Tabella 3.7:** Confronto degli iperparametri per il seed *meander02* prima e dopo l'*augmentation*

	<i>meander02</i>	<i>meander02</i> aumentato
<b>Training set (campioni)</b>	96	1536
<b>Batch size</b>	8	8
<b>Ottimizzatore</b>	Adam	Adam
<b>Learning rate (D e G)</b>	0.00002	0.0001
<b>Decay rate (D e G)</b>	0.9	0.5
<b>Funzione di costo</b>	Binary Cross Entropy	Binary Cross Entropy
<b>Epochs</b>	1000	100

## 3.4 Generazione condizionale delle immagini

Lo step finale, che conclude il lavoro svolto per questo elaborato, è quello di generare immagini sintetiche condizionali ad una specifica classe in base all'etichetta fornita, sfruttando così l'architettura CGAN scelta all'inizio del lavoro; a questo scopo, è stato implementato uno script che genera un set di immagini appartenenti alle classi *Healthy* o *Patient* in base alla scelta dell'utente.

Il listato 3.3 riporta la funzione principale dello script, responsabile della generazione delle immagini grazie al metodo *predict()* del modello generatore precedentemente caricato.

```

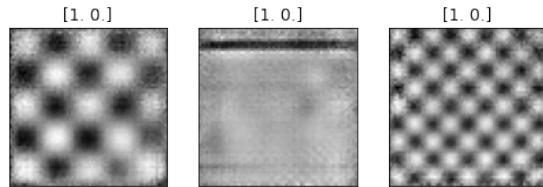
1 noise = tf.random.normal(shape=(1, latent_dim))
2 noise = tf.repeat(noise, repeats=num_images)
3 noise = tf.reshape(noise, (num_images, latent_dim))
4
5 def generate_images_and_labels(conditional_class):
6     class_label = tf.keras.utils.to_categorical([conditional_class], num_classes)
7     class_label = tf.cast(class_label, tf.float32)
8     class_label = tf.repeat(class_label, num_images, axis=0)
9     noise_and_labels = tf.concat([noise, class_label], 1)
10    synth_images = trained_generator.predict(noise_and_labels)
11    return synth_images, class_label
12
13 synth_images, labels = generate_images_and_labels(conditional_class)

```

**Listato 3.3:** Script per la generazione di immagini sintetiche

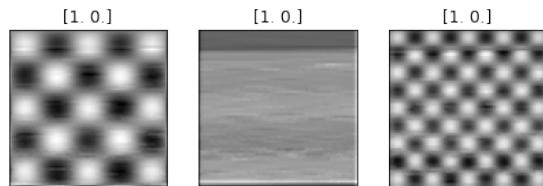
La funzione campiona del rumore da una distribuzione gaussiana e lo concatena alle etichette delle classi dopo averle convertite in formato *one-hot-vector*; successivamente, utilizzando il metodo *predict()*, dal rumore campionato viene generata un'immagine sintetica condizionale alla classe corrispondente all'etichetta.

In figura 3.8 riportiamo un esempio di immagini generate in modo condizionale, appartenenti alla classe *Healthy*, dal generatore addestrato su *meander02* aumentato dopo 100 *epochs*.



**Figura 3.8:** Esempio di immagini sintetiche generate per la classe *Healthy*

Per evidenziare meglio la qualità delle immagini sintetiche prodotte da questo modello, riportiamo in figura 3.9 un esempio di campioni presi direttamente dal dataset originale non aumentato.

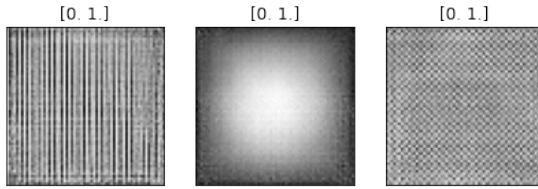


**Figura 3.9:** Esempio di immagini provenienti dal dataset di partenza per la classe *Healthy*

È importante notare che le *features* delle immagini sintetiche sono dovute alla presenza, soprattutto nel dataset dopo l'*augmentation*, di un maggior numero di campioni che presentano quelle specifiche caratteristiche (per esempio quadrati di dimensioni variabili, righe et cetera).

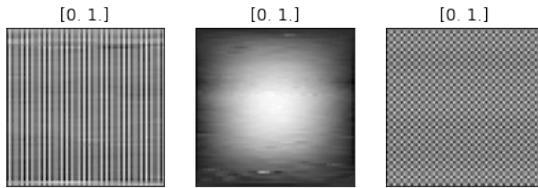
L'operazione di generazione condizionale è ripetuta anche per la classe *Patient*: grazie all'architettura CGAN, infatti, è possibile scegliere di generare immagini appartenenti ad una classe specifica e non solo casualmente come avviene per la rete GAN classica.

In figura 3.10 ne è riportato un esempio.



**Figura 3.10:** Esempio di immagini sintetiche generate per la classe *Patient*

Confrontiamo anche le immagini sintetiche appartenenti alla classe *Patient* con i campioni del dataset originale appartenenti alla stessa classe.



**Figura 3.11:** Esempio di immagini provenienti dal dataset di partenza per la classe *Patient*

### 3.4.1 Valutazione qualitativa finale

Dal confronto tra le immagini generate dal modello addestrato sul dataset originale, mostrate nel paragrafo 3.2.2, e quelle generate dopo il processo di *data augmentation* in figura 3.8 e 3.10, è evidente che il generatore addestrato sul dataset aumentato da 93 a 1536 campioni per il training set produce immagini sintetiche migliori; pur non essendo perfette e presentando ancora del rumore, le immagini sono obiettivamente più vicine ai campioni del training set iniziale mostrato in figura 3.9 e 3.11.

Un altro risultato importante da evidenziare è che tra i campioni generati non si notano più segni di *mode collapse*, in quanto le immagini prodotte dal modello finale presentano caratteristiche diverse tra loro; ciò rappresenta una migliore capacità di generalizzare da parte del modello generatore, dovuta ad un migliore apprendimento della distribuzione dei dati di training.

# Conclusioni

Il lavoro svolto presentato in questo elaborato ha avuto come obiettivo lo sviluppo di un'applicazione di Machine Learning basata su un'architettura CGAN; in particolare, lo scopo del lavoro è stato utilizzare una rete CGAN su un dataset composto da un numero molto limitato di campioni, ottenuti dalla conversione in immagini di segnali provenienti dai dati di scrittura di pazienti affetti da Malattia di Parkinson, al fine di valutare la qualità del training e la possibilità di effettuare un processo di *data augmentation* per rendere il dataset bilanciato.

Possiamo quindi distinguere il lavoro in due parti:

- Ottimizzazione dei modelli ed addestramento della rete CGAN sul dataset originale, in cui l'obiettivo è stato di individuare gli iperparametri ottimali al fine di rendere il training stabile e convergente su un numero sufficiente di *epochs*.
- Processo di *data augmentation* sul dataset tramite i metodi di TensorFlow, ottimizzazione ed addestramento della rete sul dataset aumentato, al fine di dimostrare che la qualità delle immagini generate dal modello addestrato dipende non solo dai parametri, ma soprattutto dalla natura e dalle dimensioni del dataset di partenza.

Durante la prima parte del lavoro è stata raggiunta una valutazione quantitativa buona: l'andamento del training, nello specifico delle funzioni di costo dei modelli discriminatore e generatore e delle metriche come *accuracy* e *MSE*, sono in linea con un training considerato stabile e convergente ad un range di valori soddisfacente. La valutazione qualitativa effettuata tramite ispezione visiva delle immagini generate, però, ha richiesto un ulteriore step, sviluppato nella seconda parte.

Per comprendere meglio i motivi della bassa qualità delle immagini generate dal modello nella prima parte, è stato effettuato un processo di *data augmentation*

sul dataset iniziale tramite trasformazioni casuali delle immagini, come rotazione e/o *flipping*; grazie ad esso, il seed sul quale sono stati effettuati gli esperimenti è passato da 96 a 1536 campioni per il training set. Il numero più alto di *samples* ha richiesto un *fine tuning* aggiuntivo del *learning rate* dei modelli, portando ad una valutazione qualitativa buona delle immagini sintetiche generate.

I risultati ottenuti durante il corso del lavoro portano ad una conclusione netta: il dataset iniziale presenta un numero di campioni troppo limitato per essere utilizzato da una GAN, nel nostro caso condizionale (CGAN). L’addestramento sul dataset originale, infatti, porta alla generazione di immagini sintetiche non adatte ad un processo di *data augmentation* della classe *Patient*. Il successivo aumento del numero di campioni del dataset, effettuato tramite trasformazioni casuali degli stessi ed utilizzando gli stessi modelli, ha portato invece ad immagini qualitativamente buone, confermando la necessità di utilizzare un dataset di dimensioni molto maggiori per ottenere un corretto addestramento di una CGAN.

Le conclusioni tratte portano naturalmente alla valutazione di altre strade per raggiungere gli obiettivi posti per questo lavoro.

In primis, si potrebbe migliorare l’architettura CGAN utilizzata, per esempio utilizzando una variante della classica rete GAN denominata Wasserstein GAN (WGAN) in senso condizionale (WCGAN). Questo tipo di implementazione può migliorare la stabilità dell’addestramento e risolvere i problemi di *mode collapse*; in aggiunta, si potrebbero tracciare, oltre alle metriche classiche, delle metriche specifiche al training delle GAN, come per esempio Inception score (IS) e Fréchet Inception distance (FID).

Un’ulteriore tecnica che potrebbe migliorare i risultati dell’addestramento su un dataset limitato è rappresentata dal *Transfer Learning*; con questo metodo, infatti, un modello sviluppato ed addestrato per un task viene riutilizzato come punto di partenza o base di un modello per un secondo task. Ciò risolverebbe il problema dell’addestrare i modelli su un dataset molto piccolo, dovendo aggiungere solo i *layers* relativi al nostro specifico task, come input e output, ai modelli già addestrati.

# Bibliografia

- [1] Martin Childs. *John McCarthy: Computer scientist known as the father of AI*. Nov. 2011. URL: <https://www.independent.co.uk/news/obituaries/john-mccarthy-computer-scientist-known-as-the-father-of-ai-6255307.html>.
- [2] Ana Barragán-Montero et al. «Artificial intelligence and machine learning for medical imaging: A technology review». In: *Physica Medica* 83 (2021), pp. 242–256. ISSN: 1120-1797. DOI: <https://doi.org/10.1016/j.ejmp.2021.04.016>. URL: <https://www.sciencedirect.com/science/article/pii/S1120179721001733>.
- [3] Geert Litjens et al. «A survey on deep learning in medical image analysis». In: *Medical Image Analysis* 42 (2017), pp. 60–88. ISSN: 1361-8415. DOI: <https://doi.org/10.1016/j.media.2017.07.005>. URL: <https://www.sciencedirect.com/science/article/pii/S1361841517301135>.
- [4] Tonghe Wang et al. *Medical Imaging Synthesis using Deep Learning and its Clinical Applications: A Review*. 2020. DOI: [10.48550/ARXIV.2004.10322](https://arxiv.org/abs/2004.10322). URL: <https://arxiv.org/abs/2004.10322>.
- [5] Mayank Patwari et al. «Limited parameter denoising for low-dose X-ray computed tomography using deep reinforcement learning». In: *Medical Physics* (apr. 2022). DOI: [10.1002/mp.15643](https://doi.org/10.1002/mp.15643). URL: <https://doi.org/10.1002/mp.15643>.
- [6] Michael Copeland. *The difference between AI, Machine Learning, and deep learning?* Lug. 2016. URL: <https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/>.

## BIBLIOGRAFIA

---

- [7] C. R. Pereira et al. «Deep Learning-aided Parkinson’s Disease Diagnosis from Handwritten Dynamics». In: *Proceedings of the SIBGRAPI 2016 - Conference on Graphics, Patterns and Images*. 2016, pp. 340–346.
- [8] G. Gemito. «Handwriting Analysis for Supporting the Diagnosis of Parkinson’s Disease». Tesi di laurea mag. Università degli Studi di Salerno, 2022.
- [9] Gauri Bapat. *Let’s understand the difference between machine learning vs. Deep Learning*. Ott. 2019. URL: <https://www.linkedin.com/pulse/lets-understand-difference-between-machine-learning-vs-gauri-bapat>.
- [10] Parkinson’s Foundation. *What Is Parkinson’s?* URL: <https://www.parkinson.org/understanding-parkinsons/what-is-parkinsons> (visitato il 11/03/2022).
- [11] Cleveland Clinic. *What causes Parkinson’s disease?* URL: <https://my.clevelandclinic.org/health/diseases/8525-parkinsons-disease-an-overview> (visitato il 11/03/2022).
- [12] Dr William Ju; Maksym Shcherbina; Adel Halawa; Justin Jarovi; e Maryna Pilkiw. *Neuroscience: Canadian 3rd Edition*. University of Toronto, feb. 2022.
- [13] P.A. LeWitt. «Norepinephrine: the next therapeutics frontier for Parkinson’s disease.» In: *Translational Neurodegeneration* 1.4 (2012). DOI: [10.1186/2047-9158-1-4](https://doi.org/10.1186/2047-9158-1-4). URL: <https://translationalneurodegeneration.biomedcentral.com/track/pdf/10.1186/2047-9158-1-4.pdf>.
- [14] NHS England. *What causes the loss of nerve cells?* URL: <https://www.nhs.uk/conditions/parkinsons-disease/causes/> (visitato il 11/03/2022).
- [15] Michele Di Nuzzo. *Data Science e Machine Learning: Dai Dati alla Conoscenza*. Prima Edizione. 2021. URL: <http://www.micheledinuzzo.it/dsml/>.
- [16] Tuan D. Pham e Hong Yan. «Tensor Decomposition of Gait Dynamics in Parkinson’s Disease». In: *IEEE Transactions on Biomedical Engineering* 65.8 (2018), pp. 1820–1827. DOI: [10.1109/TBME.2017.2779884](https://doi.org/10.1109/TBME.2017.2779884).
- [17] Betul Erdogan Sakar et al. «Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings». In: *IEEE Journal of Biomedical and Health Informatics* 17.4 (2013), pp. 828–834. DOI: [10.1109/JBHI.2013.2245674](https://doi.org/10.1109/JBHI.2013.2245674).

## BIBLIOGRAFIA

---

- [18] Fermín Segovia et al. «Assisted Diagnosis of Parkinsonism Based on the Striatal Morphology». In: *International Journal of Neural Systems* 29.09 (2019). PMID: 31084232, p. 1950011. DOI: [10.1142/S0129065719500114](https://doi.org/10.1142/S0129065719500114). eprint: <https://doi.org/10.1142/S0129065719500114>. URL: <https://doi.org/10.1142/S0129065719500114>.
- [19] Digital Guide - IONOS. *Deep Learning vs Machine Learning*. Mag. 2020. URL: <https://www.ionos.it/digitalguide/online-marketing/marketing-sui-motori-di-ricerca/deep-learning-vs-machine-learning/> (visitato il 06/06/2022).
- [20] IBM Cloud Education. *Neural Networks*. Ago. 2020. URL: <https://www.ibm.com/cloud/learn/neural-networks> (visitato il 06/06/2022).
- [21] Lars Ruthotto e Eldad Haber. *An Introduction to Deep Generative Modeling*. 2021. DOI: [10.48550/ARXIV.2103.05180](https://arxiv.org/abs/2103.05180). URL: <https://arxiv.org/abs/2103.05180>.
- [22] Google Developers. *Background: What is a Generative Model?* Feb. 2021. URL: <https://developers.google.com/machine-learning/gan/generative> (visitato il 06/06/2022).
- [23] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: [10.48550/ARXIV.1406.2661](https://arxiv.org/abs/1406.2661). URL: <https://arxiv.org/abs/1406.2661>.
- [24] MathWorks. *Train Conditional Generative Adversarial Network (CGAN)*. URL: <https://www.mathworks.com/help/deeplearning/ug/train-conditional-generative-adversarial-network.html> (visitato il 06/06/2022).
- [25] Mehdi Mirza e Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. DOI: [10.48550/ARXIV.1411.1784](https://arxiv.org/abs/1411.1784). URL: <https://arxiv.org/abs/1411.1784>.
- [26] Nouman Abbasi. *What is a Conditional GAN (cGAN)?* URL: <https://www.educative.io/edpresso/what-is-a-conditional-gan-cgan> (visitato il 07/06/2022).
- [27] Python Core Team. *Python: A dynamic, open source programming language*. Python version 3.9. Python Software Foundation. 2019. URL: <https://www.python.org/>.

## BIBLIOGRAFIA

---

- [28] Sayak Paul. *Conditional GAN - Training a GAN conditioned on class labels to generate handwritten digits*. 2021. URL: [https://keras.io/examples/generative/conditional\\_gan/](https://keras.io/examples/generative/conditional_gan/) (visitato il 15/05/2022).
- [29] Yann LeCun e Corinna Cortes. «MNIST handwritten digit database». In: (2010). URL: <http://yann.lecun.com/exdb/mnist/>.
- [30] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [31] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: <https://www.wandb.com/>.
- [32] Charles R Harris et al. «Array programming with NumPy». In: *Nature* 585.7825 (2020), pp. 357–362. ISSN: 1476-4687. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [33] Wes McKinney. «Data Structures for Statistical Computing in Python». In: *Proceedings of the 9th Python in Science Conference*. A cura di Stéfan van der Walt e Jarrod Millman. 2010, pp. 51–56.
- [34] John D Hunter. «Matplotlib: A 2D graphics environment». In: *Computing in science & engineering* 9.3 (2007), p. 90.
- [35] François Chollet et al. *Keras*. <https://keras.io>. 2015.
- [36] Alec Radford, Luke Metz e Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. DOI: [10.48550/ARXIV.1511.06434](https://arxiv.org/abs/1511.06434). URL: <https://arxiv.org/abs/1511.06434>.
- [37] Kinder Chen. *ReLU Activation Function Variants*. Dic. 2021. URL: <https://kinder-chen.medium.com/relu-activation-function-variants-a02dbd3c58d7> (visitato il 03/06/2022).
- [38] Augustus Odena, Vincent Dumoulin e Chris Olah. «Deconvolution and Checkerboard Artifacts». In: *Distill* (2016). DOI: [10.23915/distill.00003](https://doi.org/10.23915/distill.00003). URL: <http://distill.pub/2016/deconv-checkerboard>.

- [39] Chaoyan Zhang et al. «SCN: A Novel Shape Classification Algorithm Based on Convolutional Neural Network». In: *Symmetry* 13.3 (2021). ISSN: 2073-8994. DOI: [10.3390/sym13030499](https://doi.org/10.3390/sym13030499). URL: <https://www.mdpi.com/2073-8994/13/3/499>.
- [40] Shangeth Rajaa. *Overfitting & Regularization*. 2021. URL: <https://shangeth.com/courses/deeplearning/2.6/> (visitato il 03/06/2022).
- [41] Tim Salimans et al. *Improved Techniques for Training GANs*. 2016. DOI: [10.48550/ARXIV.1606.03498](https://doi.org/10.48550/ARXIV.1606.03498). URL: <https://arxiv.org/abs/1606.03498>.
- [42] Ali Borji. *Pros and Cons of GAN Evaluation Measures: New Developments*. 2021. DOI: [10.48550/ARXIV.2103.09396](https://doi.org/10.48550/ARXIV.2103.09396). URL: <https://arxiv.org/abs/2103.09396>.
- [43] Jason Brownlee. *How to Identify and Diagnose GAN Failure Modes*. Lug. 2019. URL: <https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/> (visitato il 09/06/2022).
- [44] Olivier Bousquet, Roi Livni e Shay Moran. «Passing Tests without Memorizing: Two Models for Fooling Discriminators». In: *CoRR* abs/1902.03468 (2019).
- [45] Yasin Yazici et al. «Empirical Analysis of Overfitting and Mode Drop in GAN Training». In: *CoRR* abs/2006.14265 (2020).
- [46] Google Developers. *GAN Training*. Apr. 2019. URL: <https://developers.google.com/machine-learning/gan/training> (visitato il 11/06/2022).
- [47] Sharon Zhou et al. *HYPE: A Benchmark for Human Eye Perceptual Evaluation of Generative Models*. 2019. DOI: [10.48550/ARXIV.1904.01121](https://doi.org/10.48550/ARXIV.1904.01121). URL: <https://arxiv.org/abs/1904.01121>.
- [48] Ali Borji. *Pros and Cons of GAN Evaluation Measures: New Developments*. 2021. DOI: [10.48550/ARXIV.2103.09396](https://doi.org/10.48550/ARXIV.2103.09396). URL: <https://arxiv.org/abs/2103.09396>.
- [49] TensorFlow. *Data augmentation*. Feb. 2022. URL: [https://www.tensorflow.org/tutorials/images/data\\_augmentation](https://www.tensorflow.org/tutorials/images/data_augmentation) (visitato il 12/06/2022).

# Elenco delle figure

1	John McCarthy nel suo laboratorio di IA a Stanford . . . . .	1
2	Intelligenza Artificiale - <i>timeline</i> . . . . .	2
3	Machine Learning vs Deep Learning . . . . .	3
1.1	Perdita dei neuroni dopaminergici della substantia nigra di pazienti affetti da MdP . . . . .	5
1.2	Programmazione classica vs Machine Learning . . . . .	7
1.3	Intelligenza Artificiale, Machine Learning e Deep Learning . . . . .	7
1.4	Esempio di Deep Neural Network . . . . .	9
1.5	Modelli discriminativi e generativi su cifre scritte a mano . . . . .	10
1.6	Modello della Generative Adversarial Network (GAN) . . . . .	11
1.7	Generatore vs Discriminatore - Inizio dell'addestramento . . . . .	11
1.8	Generatore vs Discriminatore - Progresso dell'addestramento . . . . .	12
1.9	Generatore vs Discriminatore - Fine dell'addestramento . . . . .	12
1.10	Conditional Generative Adversarial Network (CGAN) . . . . .	13
2.1	Esempio di campioni del dataset . . . . .	17
2.2	Grafico della funzione di attivazione LeakyReLU . . . . .	20
2.3	Deconvoluzione - (a) stride=2, kernel=5; (b) stride=2, kernel=4 . .	21
2.4	Riepilogo del modello generatore . . . . .	22
2.5	Esempio di layer <i>Flatten</i> . . . . .	23
2.6	Riepilogo del modello discriminatore . . . . .	24
2.7	Grafo dei modelli generatore e discriminatore . . . . .	25
2.8	Progresso dell'addestramento della CGAN . . . . .	31
3.1	Funzioni di costo di $D$ e $G$ per il seed <i>meander02</i> prima dell'ottimizzazione . . . . .	35
3.2	Funzioni di costo di $D$ e $G$ sul training set per il seed <i>meander02</i> .	38

3.3	Funzioni di costo di $G$ su training e validation set per il seed <i>meander02</i> . . . . .	38
3.4	<i>Accuracy</i> di $D$ su training e validation set per il seed <i>meander02</i> . . . . .	39
3.5	Esempio di immagini prodotte dal generatore all' <i>epoch</i> 40 . . . . .	42
3.6	Esempio di immagini prodotte dal generatore alla fine del training . . . . .	42
3.7	Esempio di immagini prodotte da $G$ addestrato sul seed <i>meander02</i> aumentato . . . . .	44
3.8	Esempio di immagini sintetiche generate per la classe <i>Healthy</i> . . . . .	46
3.9	Esempio di immagini provenienti dal dataset di partenza per la classe <i>Healthy</i> . . . . .	46
3.10	Esempio di immagini sintetiche generate per la classe <i>Patient</i> . . . . .	47
3.11	Esempio di immagini provenienti dal dataset di partenza per la classe <i>Patient</i> . . . . .	47

# Elenco delle tabelle

2.1	Valori finali degli iperparametri . . . . .	28
3.1	Esempio di parametri scelti durante la fase di <i>tuning</i> . . . . .	34
3.2	Suddivisione del dataset per ogni seed . . . . .	36
3.3	Costi e metriche registrate nella fase di training per il task <i>meander</i> all' <i>epoch</i> 1000 . . . . .	36
3.4	Costi e metriche registrate nella fase di training per il task <i>spiral</i> all' <i>epoch</i> 1000 . . . . .	37
3.5	Costi e metriche registrate nella fase di test per il task <i>meander</i> . .	40
3.6	Costi e metriche registrate nella fase di test per il task <i>spiral</i> . . .	40
3.7	Confronto degli iperparametri per il seed <i>meander02</i> prima e dopo l' <i>augmentation</i> . . . . .	45

# Elenco dei listati

2.1	Importazione delle librerie necessarie . . . . .	15
2.2	Caricamento del training set . . . . .	16
2.3	Verifica delle shapes . . . . .	17
2.4	Visualizzazione delle immagini del training set . . . . .	17
2.5	Normalizzazione del training set . . . . .	18
2.6	Custom callback per la generazione di immagini durante il training	30
2.7	Configurazione del modello ed inizio dell'addestramento . . . . .	31
2.8	Valutazione del modello sul test set . . . . .	32
3.1	Esempio di trasformazioni applicate al dataset per l' <i>augmentation</i> .	43
3.2	Creazione di un nuovo training set aumentato tramite trasformazioni	43
3.3	Script per la generazione di immagini sintetiche . . . . .	45