

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

---

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E SVILUPPO DI UN  
FRAMEWORK PER LA VALUTAZIONE DELLE  
PRESTAZIONI DI SISTEMI DI DEEPPFAKE  
DETECTION DI IMMAGINI

*Elaborato in*  
DEEP LEARNING

*Relatore*  
Prof. MATTEO FERRARA

*Presentata da*  
SERAFINO PANDOLFINI

*Correlatore*  
Dott. LORENZO PELLEGRINI

Anno Accademico 2023 – 2024



*Ci sono poche cose al mondo veramente inspiegabili e, spesso, le si ritiene tali solo a causa dei limiti della propria conoscenza.*

*Natsu Hyuuga*



# Indice

<b>Introduzione</b>	<b>vii</b>
<b>1 Modelli generativi</b>	<b>1</b>
1.1 Origine e definizione dei modelli generativi . . . . .	1
1.2 Generative Adversarial Network . . . . .	1
1.2.1 Conditional GAN . . . . .	2
1.2.2 Deep Convolutional GAN . . . . .	3
1.2.3 Wasserstein GAN . . . . .	4
1.2.4 Pix2Pix . . . . .	4
1.2.5 CycleGAN . . . . .	5
1.2.6 StyleGAN . . . . .	6
1.3 Autoencoder . . . . .	8
1.3.1 Variational Autoencoder . . . . .	9
1.3.2 Adversarial Latent Autoencoder . . . . .	10
1.4 Diffusion Models . . . . .	12
1.4.1 Denoising Diffusion Probabilistic Models . . . . .	12
1.4.2 Latent Diffusion Models . . . . .	13
1.5 Timeline dei modelli generativi . . . . .	16
<b>2 Riconoscimento di immagini sintetiche</b>	<b>21</b>
2.1 Riconoscimento di immagini sintetiche . . . . .	21
2.1.1 Artefatti Visivi . . . . .	22
2.1.2 Artefatti di compressione . . . . .	23
2.1.3 Noise Fingerprint . . . . .	24
2.2 Modelli per il riconoscimento di immagini sintetiche . . . . .	25
2.2.1 Problemi nella classificazione delle immagini generate . . . . .	25
2.2.2 CLIP . . . . .	26
2.2.3 DINOv2 . . . . .	28
2.2.4 Janus . . . . .	30
<b>3 Benchmark per la valutazione di modelli di classificazione</b>	<b>33</b>
3.1 Problemi e limitazioni della valutazione dei modelli di classificazione . . . . .	33
3.2 Soluzione proposta . . . . .	34
3.3 Caratteristiche del benchmark . . . . .	34
3.3.1 Caratteristiche funzionali . . . . .	34
3.3.2 Considerazioni sulla qualità del database . . . . .	35
3.3.3 Modalità di addestramento previste . . . . .	36
3.3.4 Caratteristiche tecniche . . . . .	36
3.4 Architettura del benchmark . . . . .	37

3.4.1 Database . . . . .	37
3.4.2 Configurazione, addestramento e valutazione dei modelli . . . . .	49
<b>4 Implementazione</b>	<b>53</b>
4.1 Costruzione del database . . . . .	53
4.1.1 Download dataset pubblici . . . . .	54
4.1.2 Costruzione dataset sintetici parziali . . . . .	55
4.1.3 Costruzione dataset reale . . . . .	65
4.2 Configurazione, pre-processing e Modelli . . . . .	68
4.2.1 Configurazione degli iperparametri . . . . .	68
4.2.2 Pipeline e tecniche di data augmentation . . . . .	72
4.2.3 Costruzione del modello . . . . .	74
<b>5 Risultati e validazione dei modelli</b>	<b>77</b>
5.1 Database . . . . .	77
5.2 Metriche utilizzate . . . . .	78
5.3 Validazione dei modelli . . . . .	80
5.3.1 Segmentazione del validation set per Sliding Window . . . . .	80
5.3.2 Modelli utilizzati . . . . .	81
5.3.3 Training . . . . .	81
<b>Conclusioni</b>	<b>87</b>
<b>Ringraziamenti</b>	<b>89</b>

# Introduzione

Negli ultimi anni, l'evoluzione delle tecnologie di deep learning ha aperto la strada a innovazioni di notevole impatto in numerosi ambiti applicativi. Tra queste, la generazione automatica di contenuti digitali (deepfake) ha suscitato sia entusiasmo per le sue potenzialità creative, sia preoccupazioni legate alla manipolazione dell'informazione. La capacità di produrre immagini sempre più realistiche, infatti, pone sfide significative per la verifica dell'autenticità dei contenuti, rendendo indispensabile lo sviluppo di sistemi di deepfake detection affidabili.

La presente tesi si propone di contribuire a questo ambito di ricerca attraverso la progettazione e lo sviluppo di un sistema di benchmarking dedicato alla valutazione delle prestazioni dei sistemi di classificazione di immagini reali e sintetiche. L'obiettivo principale è quello di fornire uno strumento standardizzato che consenta di misurare, in maniera oggettiva e comparativa, l'efficacia e l'efficienza delle diverse metodologie di classificazione applicabili.

A tal fine, il lavoro di tesi è strutturato in diversi capitoli, ciascuno dei quali affronta aspetti specifici del problema.

Nel Capitolo 1 vengono presentate le principali categorie di modelli generativi come Generative Adversarial Networks, Autoencoder e Diffusion Model. Vengono analizzati il loro funzionamento e il loro utilizzo nella creazione di immagini, evidenziando i punti di forza e le limitazioni di ciascuna metodologia, focalizzandosi sulla progressione nella qualità dei risultati con l'evolvere della tecnologia.

Nel Capitolo 2 si concentra sulle tecniche di rilevamento delle immagini sintetiche. Vengono analizzate le principali caratteristiche che contraddistinguono le immagini sintetiche come la presenza di artefatti visivi, artefatti di compressione e noise fingerprint. Inoltre, vengono illustrati alcuni modelli, come CLIP, DINOv2 e Janus, e il loro possibile utilizzo per l'estrazione di feature da immagini a supporto della classificazione.

Nel Capitolo 3 viene descritto il benchmark sviluppato per la valutazione delle prestazioni dei sistemi di deepfake detection, definendone i requisiti e l'architettura. Si discute inoltre delle caratteristiche e della qualità dei dati utilizzati e delle strategie di addestramento possibili.

Nel Capitolo 4 viene fornita una panoramica dettagliata sull'implementazione del benchmark. Vengono descritte le fasi di creazione del database, la configurazione del sistema, il pre-processing delle immagini e il training dei modelli.

Nel capitolo 5 vengono presentati e discussi i risultati ottenuti sia per la composizione del database che attraverso il training dei modelli integrati nel benchmark. L'analisi si focalizza sulle metriche utilizzate per valutare i modelli e i training effettuati sui modelli integrati.



# Capitolo 1

## Modelli generativi

### 1.1 Origine e definizione dei modelli generativi

I modelli generativi rappresentano una classe di algoritmi di machine learning progettati per apprendere la distribuzione sottostante ai dati di un determinato dominio e per generare nuovi campioni provenienti dalla stessa distribuzione. Ad esempio, un modello generativo addestrato su immagini di volti umani sarà in grado, basandosi sugli esempi osservati durante la fase di addestramento, di creare volti umani sintetici che non corrispondono a nessuna persona reale.

Questi modelli trovano applicazione in una vasta gamma di campi, dalla creazione di contenuti visivi e audio alla sintesi di dati. Tra le tecniche generative più importanti, si distinguono approcci basati su probabilità esplicite, come i **Variational Autoencoder (VAE)** [52], e approcci probabilistici impliciti, come le **Generative Adversarial Networks (GAN)** [32].

### 1.2 Generative Adversarial Network

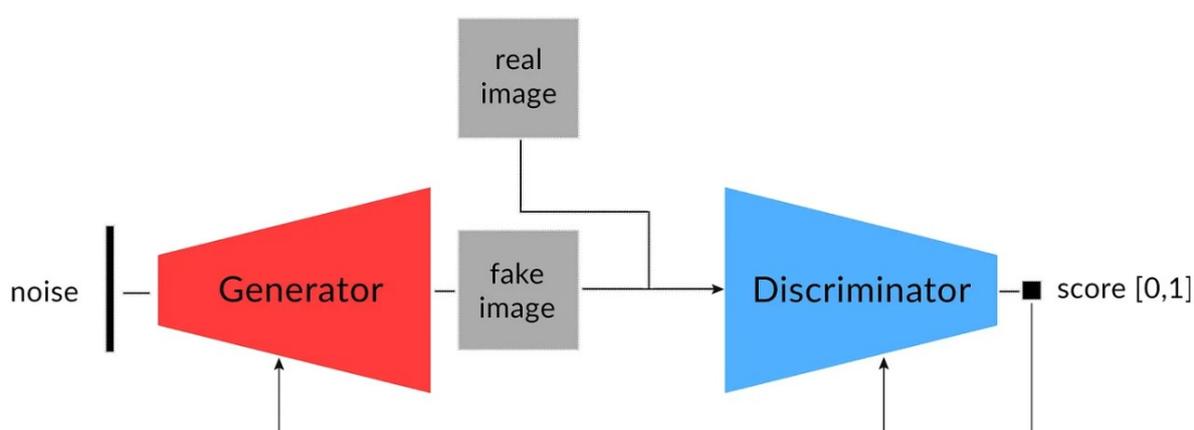


Figura 1.1: Architettura di una Generative Adversarial Network.

Il concetto e la definizione di GAN sono stati introdotti da Ian Goodfellow e colleghi nel 2014, nella pubblicazione intitolata "Generative Adversarial Nets" [32]. Questo lavoro ha proposto un framework innovativo per l'addestramento dei modelli ge-

nerativi (Figura 1.1). L'idea centrale delle GAN è quella di addestrare due modelli contemporaneamente:

- **Generatore:** tenta di creare, a partire da rumore casuale, dati sintetici realistici che possano confondersi con i dati reali del dataset.
- **Discriminatore:** cerca di distinguere tra i dati reali provenienti dal dataset e i dati generati artificialmente dal generatore.



Figura 1.2: Esempi di volti generati dal primo modello di GAN addestrato sul Toronto Face Database (TFD)[89].

I risultati ottenuti (Figura 1.2) hanno dimostrato il potenziale rivoluzionario delle GAN e hanno portato a una rapida diffusione di questa tecnologia e all'evoluzione di sue varianti come **CycleGAN**[70], **StyleGAN**[49] e **BigGAN**[11], che hanno ulteriormente migliorato la qualità e il controllo sulla generazione delle immagini.

### 1.2.1 Conditional GAN

A seguito dell'introduzione delle GAN, sono state sviluppate numerose varianti per migliorare la qualità dei risultati ottenuti. Una delle prime evoluzioni significative è rappresentata dalle **Conditional GAN (cGAN)**[61]. Questa variante introduce un meccanismo di condizionamento, aggiungendo informazioni supplementari, come etichette o altri tipi di dati, sia all'input del generatore che a quello del discriminatore (Figura 1.3). Tale approccio consente di influenzare il processo generativo in modo controllato, rendendo possibile la creazione di immagini specifiche sulla base delle informazioni fornite.

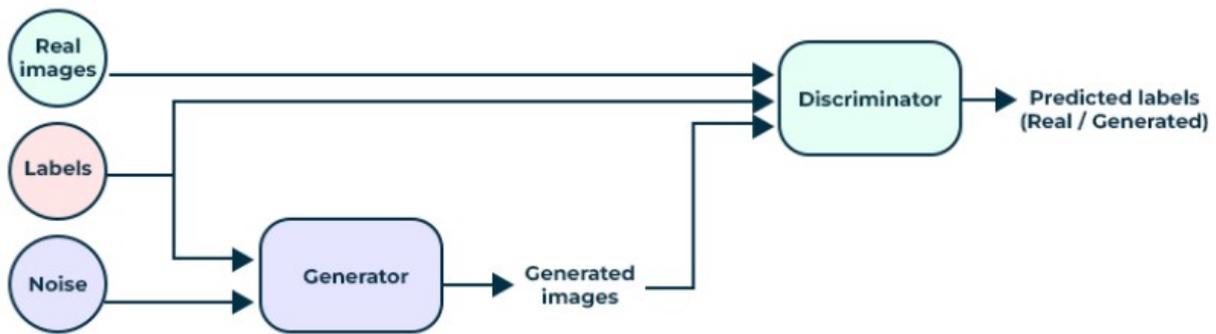


Figura 1.3: Architettura di cGAN.

## 1.2.2 Deep Convolutional GAN

Un ulteriore passo in avanti è stato compiuto con le **Deep Convolutional GAN (DCGAN)** [77], che applicano l'architettura delle Convolutional Neural Network (CNN) sia al generatore che al discriminatore, migliorando notevolmente le prestazioni delle GAN. L'architettura risultante (Figura 1.4) permette di generare immagini più realistiche e di qualità superiore, rendendo le GAN più adatte a applicazioni complesse come la generazione di immagini fotografiche (Figura 1.5).

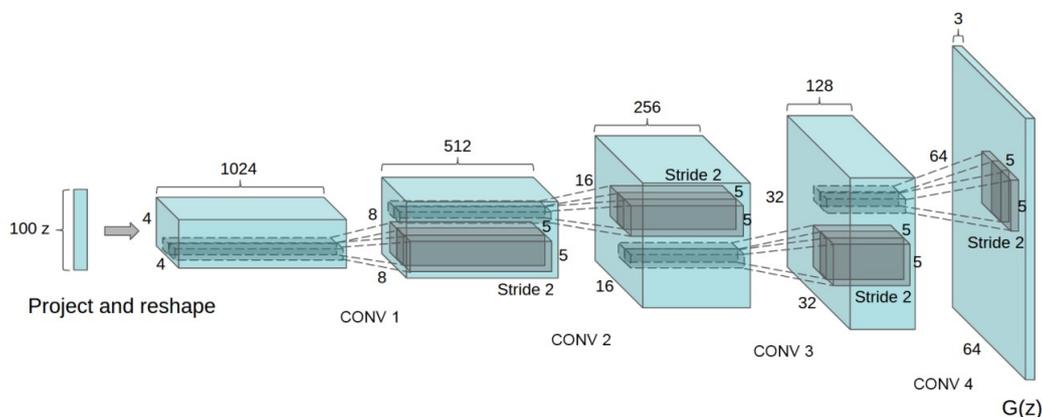


Figura 1.4: Architettura proposta per DCGAN [77].

Le DCGAN non solo hanno migliorato la qualità visiva delle immagini generate, ma hanno anche mostrato una sorprendente capacità di catturare e rappresentare le proprietà aritmetiche dei vettori nel loro spazio di rappresentazione. In particolare, i vettori ottenuti tramite le DCGAN possono essere manipolati aritmeticamente, permettendo di esplorare e combinare caratteristiche complesse delle immagini, come ad esempio la capacità di interpolare tra diverse categorie o modificare attributi specifici (come l'orientamento o l'espressione facciale). Questa proprietà ha portato a sviluppi significativi nel controllo, nella manipolazione e nella personalizzazione delle immagini generate.



Figura 1.5: Immagini di camere la letto generate tramite DCGAN addestrato sul dataset LSUN Bedrooms[101].

### 1.2.3 Wasserstein GAN

Successivamente sono stati proposti altri modelli per migliorare ulteriormente le prestazioni delle GAN, tra cui le **Wasserstein GAN (WGAN)**[5]. Questo modello introduce una nuova metrica, basata sulla distanza di Wasserstein, per valutare la qualità delle immagini generate. L'adozione di questa metrica consente di aumentare la stabilità del processo di addestramento e di ridurre problemi tipici delle GAN tradizionali, come il **mode collapse**. Il mode collapse si verifica quando il generatore si concentra sulla produzione di una sola categoria di dati che riesce a ingannare il discriminatore, generando quindi risultati ripetitivi e privi di varietà.

### 1.2.4 Pix2Pix

L'applicazione delle GAN non si limita alla generazione di immagini completamente nuove, ma si estende anche alla trasformazione di immagini esistenti, affrontando il problema dell'**image-to-image translation**. Un esempio significativo è rappresentato dal modello **Pix2Pix**[46], che introduce una variazione dell'architettura delle GAN. Questo modello utilizza coppie di immagini come input, che vengono fornite sia al generatore che al discriminatore. Questo approccio permette al generatore di apprendere una mappatura condizionata dal dominio sorgente a quello target. Utilizzando l'immagine di partenza come condizione, il generatore trasforma gli aspetti stilistici per produrre un output realistico nel dominio target, preservando al contempo la struttura spaziale e le informazioni chiave dell'immagine originale (Figura 1.6).

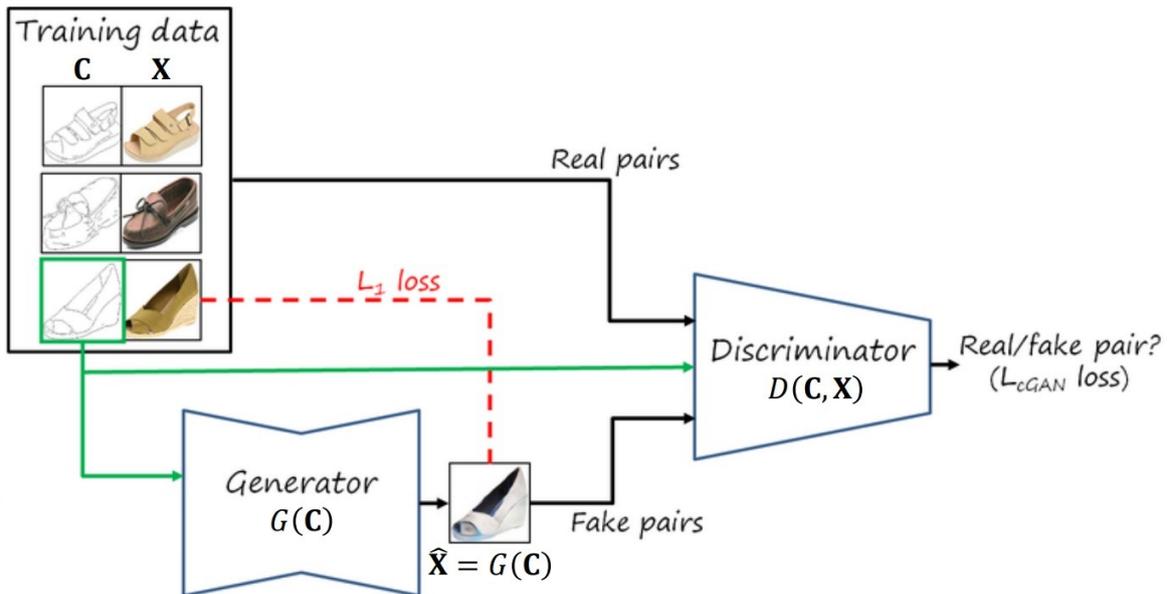


Figura 1.6: Architettura di Pix2Pix.

### 1.2.5 CycleGAN

Un'evoluzione significativa nell'ambito dell'immagine-to-immagine translation è rappresentata da **CycleGAN**[105], un modello progettato per convertire immagini tra due domini distinti utilizzando dataset non accoppiati, ovvero dataset le cui immagini dei due domini non sono presentate in coppie corrispondenti. Ad esempio, dato un dataset di immagini di mele (dominio A) e uno di arance (dominio B), CycleGAN è in grado di trasformare un'immagine del dominio A in una sua variante realistica nel dominio B e viceversa (Figura 1.7).

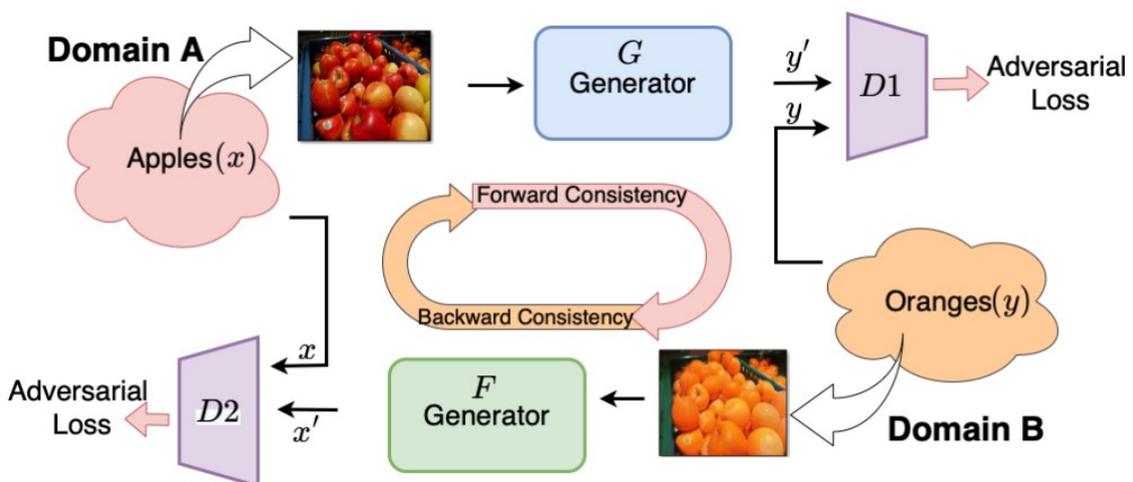


Figura 1.7: Schema architetturale di CycleGAN.

Il concetto fondamentale alla base delle CycleGAN è la **cycle-consistency**: quando un'immagine del dominio A viene trasformata nel dominio B e successivamente riportata nel dominio A, l'immagine finale dovrebbe risultare identica all'originale. Per ottenere questo risultato, il modello utilizza due generatori (uno per ogni direzione di

trasformazione tra i due domini) e due discriminatori (che valutano la qualità delle immagini generate in ciascun dominio). I generatori apprendono a produrre immagini convincenti nel dominio opposto, mentre i discriminatori si assicurano che le immagini generate siano realistiche e coerenti con il dominio di destinazione. (Figura 1.8)

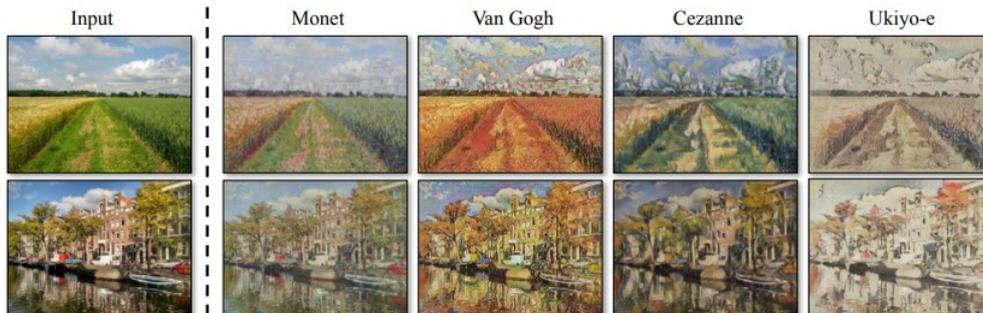


Figura 1.8: Esempio di trasferimento dello stile tramite CycleGAN da immagine allo stile di Monet, Van Gogh, Cezanne e Ukiyo-e.

## 1.2.6 StyleGAN

Tra le innovazioni principali nel campo delle GAN emerge **StyleGAN**[49], sviluppato da NVIDIA, che fornisce nuovi metodi per il controllo del contenuto delle immagini generate. Rispetto a una GAN tradizionale, l'architettura di StyleGAN prevede di utilizzare un input e mapparlo in uno spazio latente che influenza la generazione dell'immagine ad ogni operazione di convoluzione (Figura 1.9).

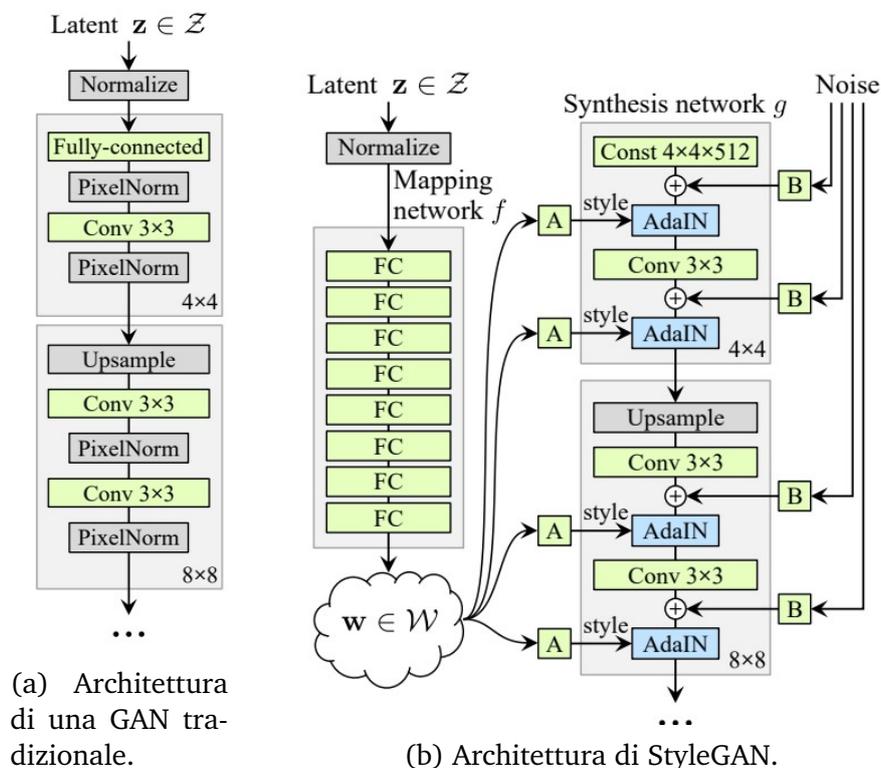


Figura 1.9: Differenze tra l'architettura utilizzata da GAN tradizionali e quella utilizzata da StyleGAN con l'impiego dello spazio latente.

StyleGAN utilizza un meccanismo di **style mixing**, che consente di iniettare spazi latenti differenti in specifici livelli della rete generativa, permettendo il controllo indipendente degli attributi globali e locali e la combinazione coerente delle proprietà stilistiche provenienti da input differenti.

La qualità delle immagini raggiunta da StyleGAN è tale da poter confondere immagini generate con immagini reali (Figura 1.10).



Figura 1.10: Esempi di immagini realizzate con StyleGAN.

Tra le architetture GAN più innovative che hanno ulteriormente migliorato le prestazioni, vi sono inoltre:

- **Projected GAN:** si distingue per un discriminatore che “proietta” le immagini generate e reali su un feature space pre-addestrato, migliorando la stabilità del training, riducendo il tempo impiegato da questa fase, e la qualità dell’immagine generata[86].
- **GANformer:** estende il concetto introdotto da integrando transformer nel generatore, ottenendo dettagli più accurati nelle immagini rispetto agli approcci precedenti[44].
- **GauGAN:** migliora il controllo semantico nella generazione, consentendo la sintesi di scene fotorealistiche a partire da descrizioni testuali e informazioni semantiche. La sua architettura ibrida permette di fondere informazioni condizionali di diversa tipologia in modo efficace (Figura 1.11), garantendo alta qualità e precisione nei risultati[71].

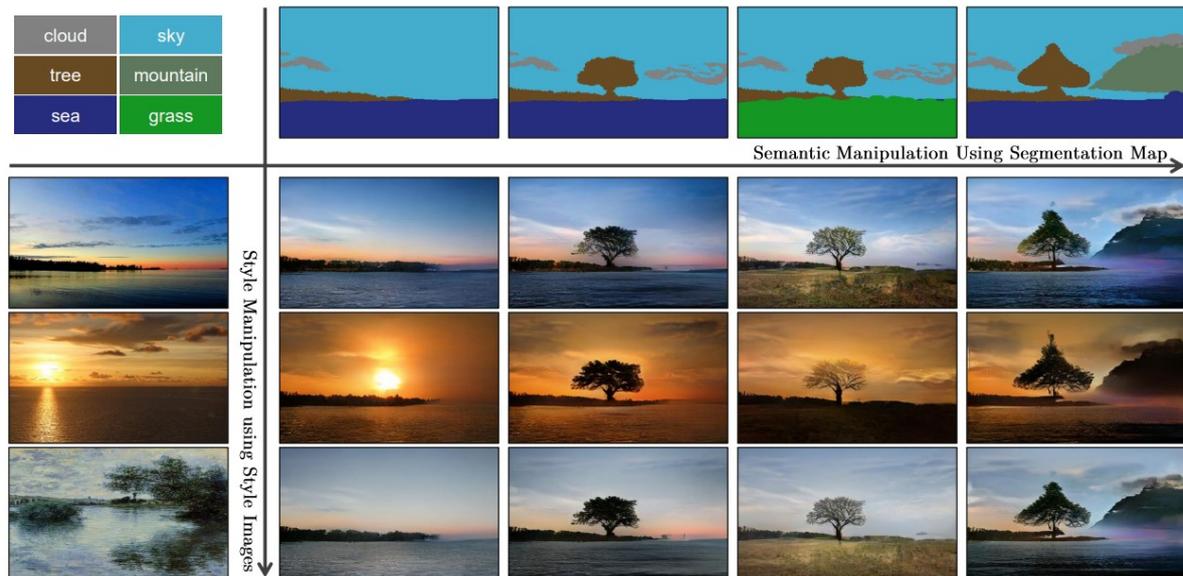


Figura 1.11: Immagini generate da GauGAN a partire da una segmentation map e da un'immagine per definirne lo stile e il contenuto.

### 1.3 Autoencoder

Oltre alle GAN, nel corso degli anni sono emersi altri modelli generativi basati su tecniche e architetture differenti, come modelli basati su autoencoder.

Gli autoencoder sono modelli in grado di apprendere come comprimere (codificare) i dati ricevuti in input in una rappresentazione latente di dimensione ridotta e ricostruirli (decodificare) in modo tale da minimizzare l'errore di ricostruzione. Un autoencoder è costituito da due componenti principali:

- **Encoder:** trasforma l'input in una sua rappresentazione nello spazio latente.
- **Decoder:** ricostruisce l'input a partire dalla rappresentazione nello spazio latente.

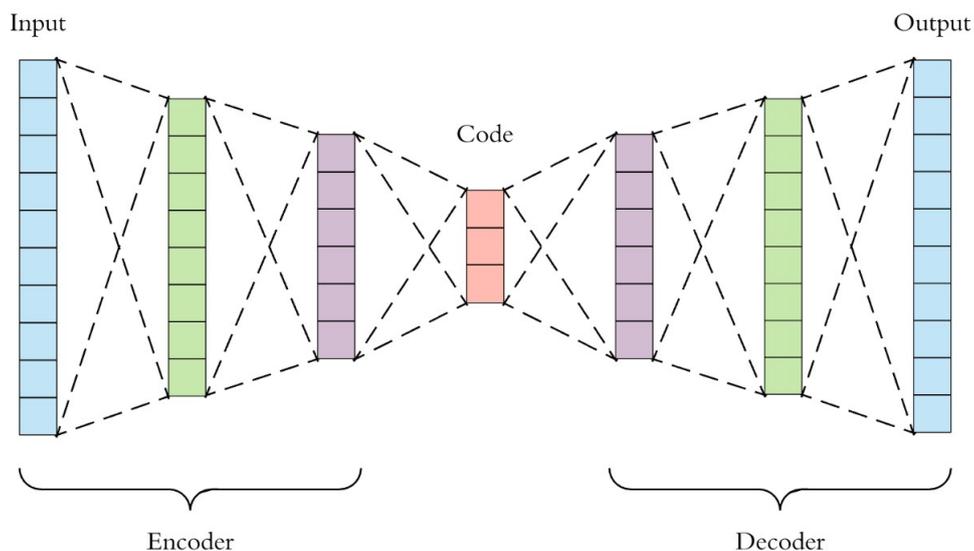


Figura 1.12: Struttura del modello di un autoencoder.

Inizialmente impiegati come alternativa alla Principal Component Analysis (PCA)[40] per la riduzione della dimensionalità, gli autoencoder offrono il vantaggio di catturare trasformazioni non lineari, permettendo di estrarre caratteristiche più complesse e significative dai dati rispetto ai metodi lineari tradizionali. Come evidenziato da Hinton e Salakhutdinov in "Reducing the Dimensionality of Data with Neural Networks"[38], utilizzando architetture profonde è possibile ottenere rappresentazioni latenti che migliorano notevolmente le performance in compiti di riconoscimento, classificazione e visualizzazione dei dati, grazie alla capacità di modellare le relazioni non lineari presenti nei dati di alta dimensionalità.

### 1.3.1 Variational Autoencoder

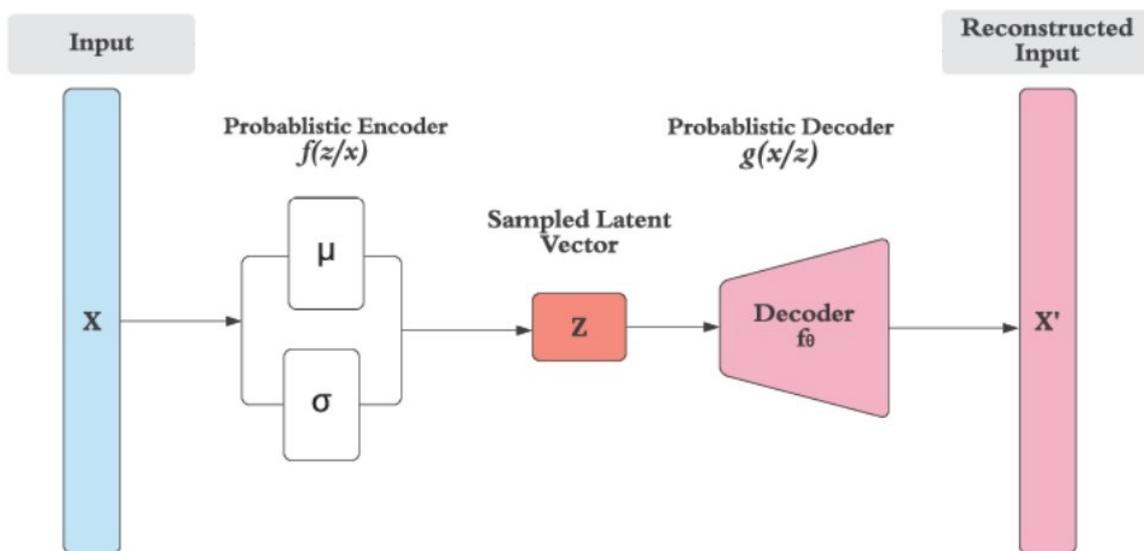


Figura 1.13: Architettura di un Variational Autoencoder.

Successivamente, il paradigma è stato esteso all'ambito dei modelli generativi, con l'introduzione del **Variational Autoencoder (VAE)**[52], un modello generativo che estende i principi degli autoencoder classici, introducendo un approccio probabilistico nello spazio latente. Invece di rappresentare ogni input come un singolo punto nello spazio latente, l'encoder di un VAE trasforma l'input in una distribuzione che cattura le caratteristiche fondamentali del dato. Durante l'addestramento il decoder impara a ricostruire l'input originale a partire da un campione prelevato da questa distribuzione. Questa organizzazione permette di generare nuovi dati: una volta addestrato, basta campionare casualmente dallo spazio latente e passare il risultato al decoder, che lo trasforma in un output realistico e coerente con i dati di addestramento (Figura 1.13).

Una variante dei VAE è rappresentata dai **Vector Quantised-Variational AutoEncoder (VQ-VAE)**[94]. Rispetto ai VAE tradizionali, i VQ-VAE non codificano i dati in uno spazio latente continuo, ma li approssimano a valori discreti, migliorando così la nitidezza delle immagini prodotte dal decoder. Questo approccio consente di ottenere una rappresentazione più compatta e di migliorare la qualità complessiva delle immagini generate.

### 1.3.2 Adversarial Latent Autoencoder

Un'altra evoluzione nell'ambito dei modelli generativi basati su autoencoder è rappresentata dagli **Adversarial Latent Autoencoder (ALAE)**[73]. A differenza dei VAE e delle loro varianti, gli ALAE combinano i principi degli autoencoder con quelli delle GAN, introducendo un meccanismo di adversarial training nello spazio latente.

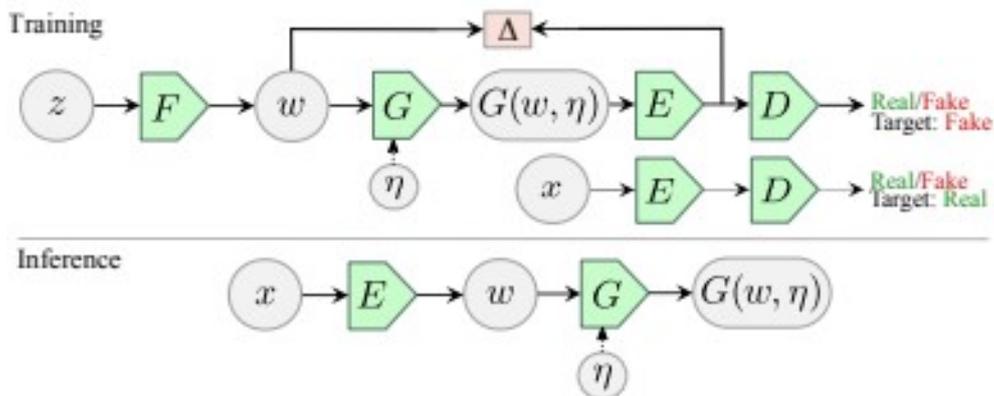


Figura 1.14: Architettura di ALAE.

L'architettura di ALAE (Figura 1.14) utilizza la struttura base delle GAN, ovvero generatore e discriminatore, scomponendoli rispettivamente in due parti: la prima per convertire l'immagine in input nella sua rappresentazione latente (F ed E), la seconda per generare (G) o discriminare (D) l'immagine a partire dalla rappresentazione latente. Per il corretto funzionamento del modello, gli spazi latenti in cui sono mappate le immagini sia durante la fase di generazione che di discriminazione devono essere quanto più simili possibili. La discrepanza tra le due distribuzioni ( $\Delta$ ) viene minimizzata dalla funzione loss utilizzata.

L'approccio e il modello adottato consentono agli ALAE di ottenere risultati comparabili alle migliori architetture GAN, mantenendo al contempo la capacità di inversione e ricostruzione tipica degli autoencoder.

Estendendo il principio di ALAE, è stato sviluppato un modello in grado di replicare le caratteristiche di StyleGAN, denominato **StyleALAE** (Figura 1.15), dove a differenza del vettore latente di StyleGAN viene utilizzato lo spazio latente dell'encoder.

Un vantaggio chiave di StyleALAE è la capacità di manipolare separatamente le caratteristiche globali e locali dell'immagine grazie alla sintesi per livelli dell'encoder utilizzato, in modo simile a StyleGAN. Tuttavia, a differenza di quest'ultimo, l'uso di un encoder consente di sfruttare informazioni più ricche per generare immagini più fedeli ai dati di training, migliorando la qualità delle ricostruzioni (Figura 1.16).

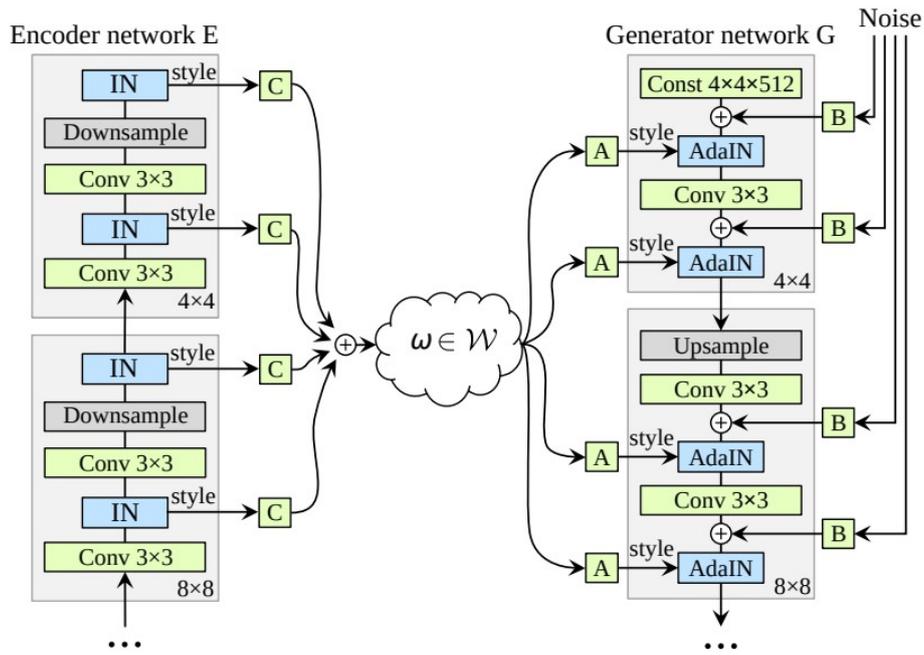


Figura 1.15: Architettura di StyleALAE (encoder e generatore).



(a) Immagini di volti generati.



(b) Ricostruzione su immagini non note al modello.

Figura 1.16: Risultati del modello StyleALAE addestrato sul dataset FFHQ[49].

## 1.4 Diffusion Models

I Diffusion Models, introdotti nel lavoro di Sohl-Dickstein e colleghi, "Deep Unsupervised Learning using Nonequilibrium Thermodynamics"[88], si basano su un processo probabilistico che trasformano progressivamente un campione di dati in rumore, mediante una sequenza di passi deterministici che simulano un processo stocastico. Questo processo di diffusione rimuove gradualmente le strutture nei dati originali, con l'obiettivo di ottenere una distribuzione semplice, come una distribuzione gaussiana. La novità dei Diffusion Models è che, anziché generare dati direttamente dalla distribuzione, viene appreso il processo inverso che, partendo dal rumore, ricostruisce i dati originali passo dopo passo (Figura 1.17). In questo modo, si sfrutta la reversibilità delle dinamiche stocastiche per offrire una solida architettura generativa, ponendo le basi per l'evoluzione dei modelli generativi.

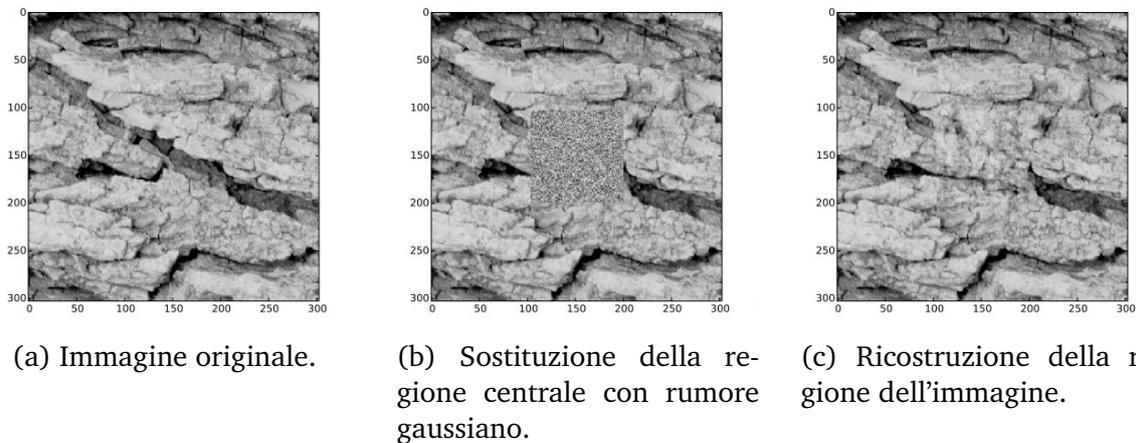


Figura 1.17: Esempio di inpainting realizzato tramite Diffusion Models.

### 1.4.1 Denoising Diffusion Probabilistic Models

Una prima evoluzione dei Diffusion Models è rappresentata dai **Denoising Diffusion Probabilistic Models (DDPM)**[39], che si basano su un processo probabilistico suddiviso in due fasi: una di diffusione e una di denoising.

- **Fase di diffusione:** durante questa fase, il modello trasforma un'immagine di input in puro rumore tramite una sequenza di passi deterministici.
- **Fase di denoising:** questa fase inverte il processo di diffusione, ricostruendo iterativamente l'immagine a partire dal rumore (Figura 1.18).



Figura 1.18: Iterazioni della fase di denoising di DDPM.

La funzione loss utilizzata calcola la differenza tra il rumore effettivamente iniettato durante la fase di diffusione e quello stimato dal modello in ciascun passaggio, in modo che minimizzando tale errore il modello impari a rimuovere progressivamente il rumore per ricostruire l'immagine originale. Questo approccio semplifica l'ottimizzazione e consente al modello di apprendere efficacemente il processo inverso.

I risultati prodotti dai DDPM sono di qualità elevata (Figura 1.19), comparabili a quelli dei modelli GAN più avanzati, ma senza soffrire di problematiche come il **mode collapse**. Tuttavia, la qualità del risultato dipende strettamente dal numero di iterazioni effettuate durante la fase di denoising, rendendo il modello relativamente lento nel produrre immagini di alta qualità.

## 1.4.2 Latent Diffusion Models

Un'evoluzione diretta dei DDPM è rappresentata dai **Latent Diffusion Models (LDM)**[83], che combinano l'architettura di un **Variational Autoencoder (VAE)** con le fasi di diffusione e denoising tipiche dei DDPM (Figura 1.20).

Il processo degli LDM prevede tre fasi principali:

1. Codifica nello spazio latente: un'immagine originale viene mappata, tramite l'encoder del VAE, in uno spazio latente a dimensione ridotta.
2. Diffusione e denoising nello spazio latente: le fasi di diffusione e denoising vengono applicate direttamente su questa rappresentazione latente, riducendo così la complessità del problema rispetto ai DDPM, che operano nello spazio delle immagini.
3. Decodifica: il decoder del VAE ripristina l'immagine finale a partire dalla rappresentazione latente generata.

Questa strategia consente di ridurre i requisiti computazionali e di ridurre il tempo di generazione, mantenendo al contempo un'elevata qualità dei risultati.

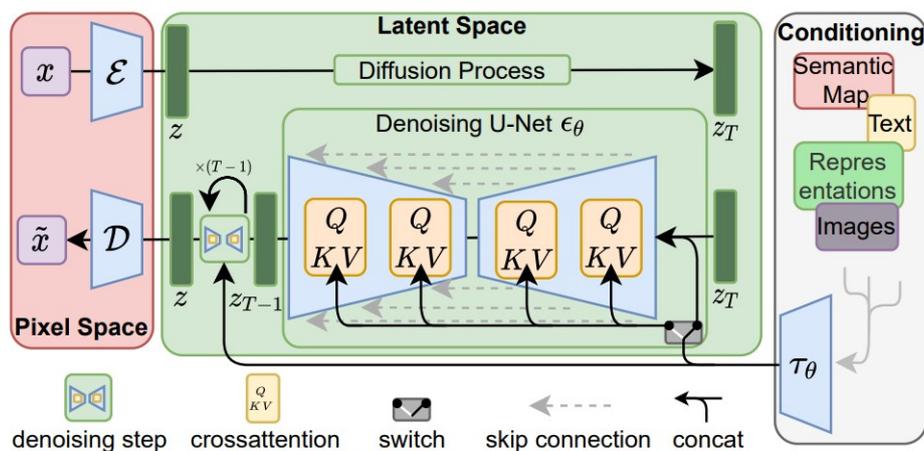


Figura 1.20: Architettura di LDM: l'immagine originale  $x$  viene processata dall'encoder, dal processo di diffusione (che trasforma la rappresentazione latente in rumore), dalla fase di denoising (che può includere informazioni per il condizionamento dell'immagine) e infine dalla fase di decoding per ottenere l'immagine finale  $\tilde{x}$ .



Figura 1.19: Esempi di immagini realizzate con DDPM addestrato sul dataset LSUN[101].

Gli LDM possono essere condizionati tramite informazioni esterne durante la fase di denoising. Questo è possibile combinando la rappresentazione latente con segnali esterni, come descrizioni testuali, maschere o etichette, per influenzare il contenuto e le caratteristiche dell'immagine generata.

Un'importante applicazione degli LDM è l'**image inpainting**, che consente di completare regioni mancanti o danneggiate di un'immagine. Questo approccio può essere utilizzato per riparare immagini deteriorate o per rimuovere e sostituire contenuti indesiderati. I risultati ottenuti con LDM in questo ambito si sono dimostrati eccellenti (Figura 1.21), al punto che in alcuni test osservatori umani hanno preferito i risultati generati dagli LDM rispetto a quelli prodotti da modelli progettati specificamente per questo obiettivo.

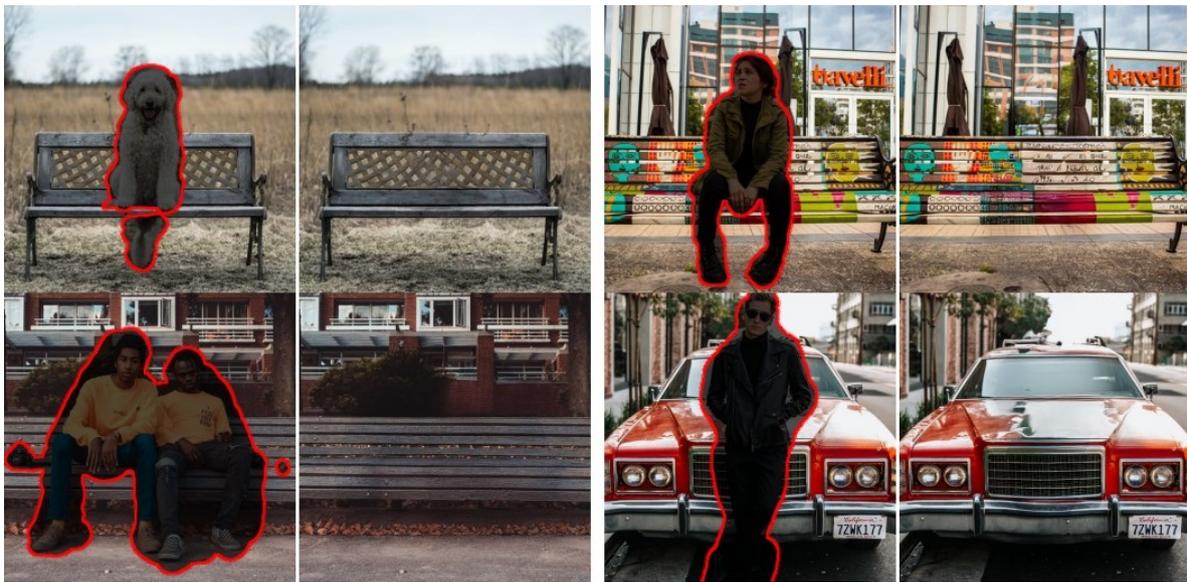


Figura 1.21: Esempi di image inpainting utilizzando LDM.

Oltre ai DDPM e agli LDM, sono stati proposti altri modelli che apportano novità significative per la qualità delle immagini e i tempi di generazione:

- **DALL-E V2**: un'evoluzione del modello DALL-E, che combina il paradigma diffusion con tecniche di super-risoluzione e un controllo più fine delle caratteristiche visive, migliorando la coerenza tra testo e immagine[68].
- **Stable Diffusion 2.0**: una versione aggiornata dei modelli Stable Diffusion in cui il processo di generazione viene eseguito interamente nello spazio latente. Grazie a un VAE ottimizzato, le immagini vengono compresse in rappresentazioni compatte, permettendo un denoising più efficiente che si traduce in immagini di qualità superiore e tempi di generazione ridotti (Figura 1.22).
- **Diffusion GAN**: un approccio ibrido che combina la robustezza e la qualità della generazione tipica dei diffusion models alla velocità di generazione delle GAN[96].



Figura 1.22: Esempi di immagini con Stable Diffusion 2.0.

## 1.5 Timeline dei modelli generativi

Di seguito è riportata la timeline dei principali modelli generativi rilasciati nel corso degli ultimi anni associati alla rispettiva categoria.

Data Rilascio	Nome Modello	Categoria
2016-12-10	StackGAN[104]	GAN
2017-03-30	CycleGAN[105]	GAN
2017-10-27	ProGAN[47]	GAN
2017-11-02	VQ-VAE[94]	Autoencoder
2017-11-24	StarGAN V1[18]	GAN
2017-11-29	AttGAN[37]	GAN
2018-06-10	SN-PatchGAN[102]	GAN

2018-09-28	BigGAN[11]	GAN
2018-12-12	StyleGAN V1[49]	GAN
2019-01-29	RelGAN[65]	GAN
2019-03-18	GauGAN[71]	GAN
2019-06-02	VQ-VAE-2[79]	Autoencoder
2019-12-03	StyleGAN V2[50]	GAN
2019-12-04	StarGAN V2[18]	GAN
2020-06-19	DDPM[39]	Diffusion
2020-11-27	CIPS[4]	GAN
2020-12-17	VQ-GAN[28]	GAN + Autoencoder
2021-03-01	GANformer[44]	GAN
2021-05-11	ADM[25]	Diffusion
2021-06-23	StyleGAN V3[48]	GAN
2021-11-01	Projected GAN[86]	GAN
2021-11-10	Palette[84]	Diffusion
2021-11-17	GANformer2[43]	GAN
2021-11-29	VQ-Diffusion[35]	Diffusion
2021-12-09	GauGAN2[41]	GAN
2021-12-15	Eg3D[14]	GAN
2021-12-15	Denoising Diffusion GAN[100]	Diffusion
2021-12-20	GLIDE[64]	Diffusion
2021-12-20	Latent Diffusion[83]	Diffusion
2022-02-01	MidJourney V1[93]	Diffusion
2022-04-07	DALL-E V2[68]	Diffusion
2022-04-12	MidJourney V2[93]	Diffusion
2022-05-23	Imagen V1[85]	Diffusion
2022-06-05	Diffusion GAN[96]	GAN + Diffusion
2022-07-25	MidJourney V3[93]	Diffusion
2022-08-18	Stable Diffusion 1.4[20]	Diffusion
2022-10-20	Stable Diffusion 1.5[20]	Diffusion
2022-11-05	MidJourney V4[93]	Diffusion
2022-11-24	Stable Diffusion 2.0[20]	Diffusion
2022-12-01	Stable Diffusion 2.1[20]	Diffusion
2022-12-08	Wukong[34]	Diffusion
2023-03-15	MidJourney V5[93]	Diffusion
2023-03-21	Firefly 1[45]	Diffusion
2023-04-26	DeepFloyd-IF[3]	Diffusion
2023-07-26	Stable Diffusion XL 1.0[20]	Diffusion
2023-09-20	DALL-E V3[68]	Diffusion
2023-10-06	Latent Consistency Models[58]	Diffusion

2023-10-10	Firefly 2[45]	Diffusion
2023-11-28	Stable Diffusion XL Turbo[20]	Diffusion
2023-12-21	MidJourney V6[93]	Diffusion
2024-02-01	Imagen V2[19]	Diffusion
2024-02-22	Stable Diffusion 3.0[20]	Diffusion
2024-04-23	Firefly 3[45]	Diffusion
2024-05-14	Imagen V3[19]	Diffusion
2024-08-01	Flux.1 [53]	Diffusion

Tabella 1.1: Elenco dei principali modelli generativi rilasciati dalla fine del 2016 ad agosto 2024.

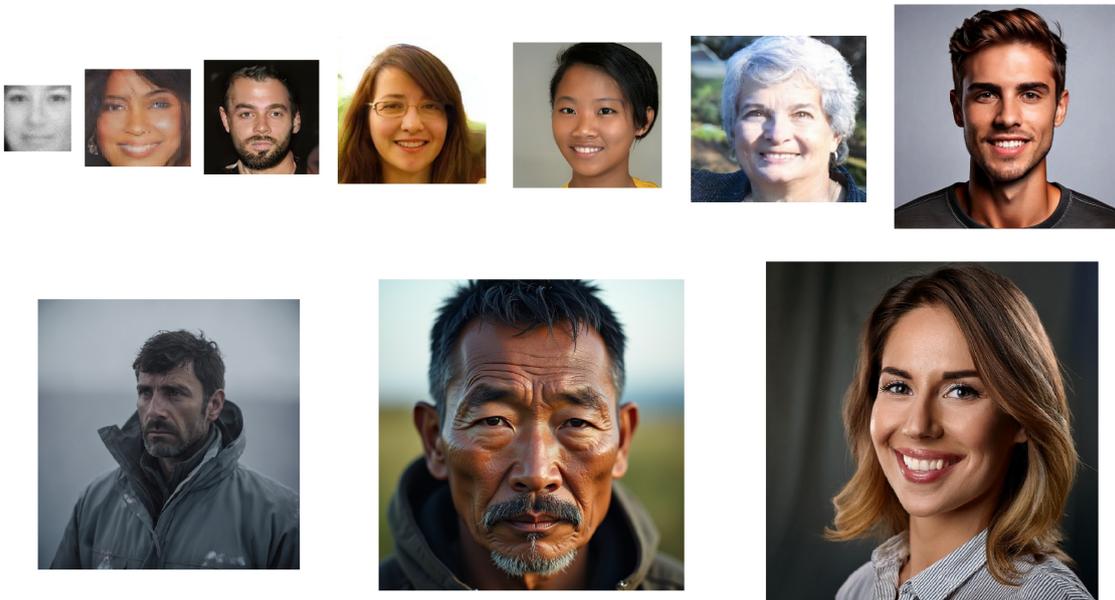


Figura 1.23: Evoluzione della qualità delle immagini generate nel corso negli anni. Immagini generate da: GAN[32], DCGAN[77], ProGAN[47], StyleGAN[49], StyleGAN3[48], Latent Diffusion[83], DALL-E V3[68], Stable Diffusion XL[20], Flux.1[53] e Firefly3[45]

Con il passare del tempo, oltre a un costante miglioramento nei risultati dei modelli generativi (Figura 1.23), si osserva una transizione dai modelli basati su GAN a quelli basati su diffusion, una scelta determinata da una serie di fattori che rendono questi ultimi preferibili[25][63].

- **Stabilità nell'addestramento:** le architetture basate su GAN sono caratterizzate dalla competizione tra generatore e discriminatore, portando spesso a problemi di convergenza e al fenomeno del mode collapse. Al contrario, il processo di denoising dei diffusion models garantisce una maggiore stabilità durante l'addestramento e permette di ottenere modelli più robusti.

- **Qualità delle immagini e copertura della distribuzione:** nonostante richiedano un numero elevato di passaggi per generare un'immagine, i diffusion models sono in grado di produrre risultati qualitativamente superiori, di coprire in modo più completo la distribuzione dei dati di addestramento e di catturarne la complessità.
- **Controllo e flessibilità:** rispetto alle GAN il processo di generazione dei diffusion models consente interventi mirati, permettendo di condizionare la generazione su informazioni esterne (ad esempio, testi o altri segnali) e di ottenere modifiche graduali e precise. Questa flessibilità è fondamentale per applicazioni che richiedono un elevato grado di personalizzazione e per scenari in cui è necessario guidare il processo generativo verso specifiche caratteristiche desiderate.



# Capitolo 2

## Riconoscimento di immagini sintetiche

### 2.1 Riconoscimento di immagini sintetiche

Negli ultimi anni, la diffusione dei modelli generativi, come **Stable Diffusion**[20], **Midjourney**[93] e **DALL-E**[68], ha reso la generazione di contenuti non solo visivi, ma anche audio e testuali, accessibile a un pubblico molto più vasto, anche a coloro che non possiedono competenze tecniche o artistiche avanzate. Questo ha permesso a chiunque di esplorare e generare contenuti in modo semplice e intuitivo. Sempre più contenuti vengono prodotti senza l'intervento diretto umano, portando alla creazione di una grande varietà di opere. Sebbene ciò favorisca l'innovazione, solleva anche interrogativi sull'autenticità di tali contenuti. Pur raggiungendo una qualità paragonabile a quella umana, le opere generate dall'Intelligenza Artificiale (IA) soffrono di una sostanziale mancanza di veridicità e autenticità, elementi che ne compromettono il valore artistico e la genuinità rispetto alle opere tradizionali. Allo stato dell'arte attuale, con l'evoluzione dei modelli generativi, è diventato sempre più difficile distinguere i contenuti reali da quelli sintetici (generati dall'IA), considerando la varietà e la qualità dei contenuti che questi modelli sono in grado di produrre.

In generale, i modelli basati sull'IA generativa hanno introdotto nuove vulnerabilità e opportunità per gli attacchi informatici. Tra le minacce emergenti, spiccano il **spear-phishing**, possibile grazie alla capacità di generare email altamente personalizzate e convincenti, e la diffusione di **deepfake** (Figura 2.1), che sfruttano l'alta fedeltà dei modelli generativi per diffondere disinformazione su larga scala[8].



Figura 2.1: Esempio di disinformazione creato tramite IA generativa.

Nel riconoscimento delle immagini sintetiche le tecniche si focalizzano sull'individuazione delle "impronte digitali" lasciate dai modelli generativi durante la creazione [12][91]. Tali tracce possono essere suddivise in due macro categorie: da un lato, gli artefatti visivi che sono percepibili a occhio nudo; dall'altro, le tracce che richiedono di essere estratte dall'immagine come gli artefatti di compressione o la noise fingerprint.

### 2.1.1 Artefatti Visivi

Gli **artefatti visivi** sono difetti visibili nelle immagini che rivelano la natura artificiale del contenuto generato. Questi difetti derivano dalle limitazioni intrinseche dei modelli generativi, che possono fallire nel replicare perfettamente dettagli come illuminazione, geometria o elementi complessi. Tipicamente, queste inconsistenze visive sono facilmente identificabili focalizzandosi sui dettagli dell'immagine, piuttosto che sul suo insieme. Ad esempio, osservando delle mani, si potrebbe notare un numero innaturale di dita, oppure alcuni oggetti potrebbero essere mal costruiti. In altri casi, il testo presente potrebbe essere totalmente casuale e le lettere stesse non riconoscibili come tali (Figura 2.2).

Anche se l'identificazione di artefatti visivi è attualmente una delle principali tecniche di rilevamento, con il miglioramento dei modelli generativi, è probabile che tali errori diventino sempre meno evidenti e sempre più difficili da identificare.



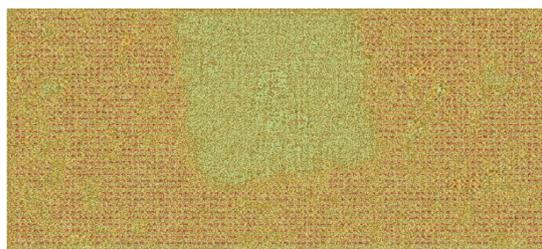
Figura 2.2: Tipici artefatti visivi prodotti dai modelli generativi: esempio di mano con un numero errato di dita (a), sedia a rotelle costruita in modo errato (b), testo non leggibile e deformato (c) e riflesso incoerente (d).

### 2.1.2 Artefatti di compressione

Negli ultimi anni l'interesse verso l'analisi forense delle immagini sintetiche è cresciuto in maniera esponenziale, in risposta alla diffusione di modelli generativi sempre più sofisticati. Questa tipologia di residui presenti nelle immagini generate, quali distribuzione dei pixel, i pattern di compressione, noise fingerprint e i metadati, richiedono di essere prima estratti dall'immagine e poi sottoposti ad analisi per individuare incoerenze o anomalie che possano indicare manipolazioni (image inpainting) o una natura artificiale dell'immagine.



(a) Immagine modificata.



(b) Compression fingerprint dell'immagine modificata.

Figura 2.3: Esempio di immagine modificata con relativa compression fingerprint.

Di questi, i pattern di compressione permettono di individuare in modo efficace l'alterazione di immagini tramite inpainting. Gli artefatti di compressione emergono a causa del processo intrinseco di quantizzazione applicato durante la compressione JPEG. In questo procedimento, l'immagine viene divisa in blocchi e trasformata al fine di diminuire i dati necessari a rappresentare l'immagine. Quando un'immagine viene modificata e poi salvata nuovamente in JPEG, le regioni alterate subiscono spesso una compressione diversa rispetto al resto dell'immagine. Queste differenze di compressione sono osservabili anche in immagini completamente generate, in quanto spesso i modelli generativi sono addestrati su immagini con diverse tipologie di compressione e producono risultati che riflettono queste compressioni a diversi livelli[60]. Recenti studi hanno dimostrato che i modelli generativi, addestrati su dataset eterogenei in termini di compressione, tendono a "ereditare" i differenti pattern di compressione presenti nelle immagini di training e, di conseguenza, a produrre immagini sintetiche che contengono combinazioni di questi pattern. Queste evidenze suggeriscono che le differenze di compressione rilevabili in un'immagine non sono solo indicatori di alterazione, ma anche una caratteristica intrinseca dei contenuti generati artificialmente, derivante dalla varietà di compressioni presenti nei dati di addestramento[75].

### 2.1.3 Noise Fingerprint

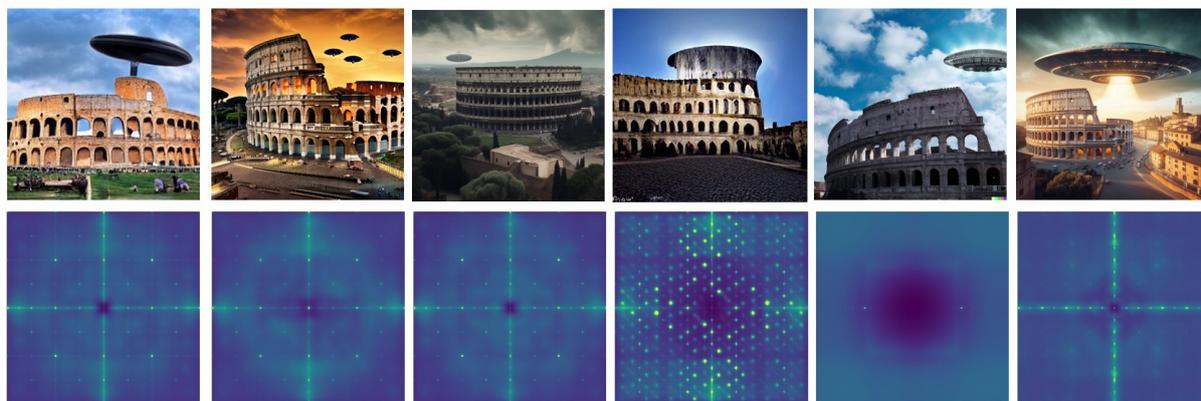


Figura 2.4: Confronto tra noise fingerprint di immagini generate da diversi modelli: Latent Diffusion[83], Stable Diffusion[20], Midjourney v5[93], DALL·E Mini[23], DALL·E V2[68], DALL·E V3[68]

Un altro approccio per il riconoscimento di immagini generate prevede l'analisi delle **noise fingerprints**, ovvero il rumore residuo che rimane dopo la generazione di un'immagine da parte di un modello. Ogni fotocamera e ogni modello generativo produce un pattern di rumore specifico, che può essere utilizzato per identificare non solo se l'immagine è reale o generata, ma anche da quale generatore è stata prodotta (figura 2.4).

Con l'avanzamento delle tecniche di identificazione, sono emersi attacchi appositamente progettati per ingannare i sistemi di rilevamento. Questi attacchi mirano a mascherare le tracce lasciate dai modelli generativi, riducendo così l'efficacia delle tecniche di riconoscimento. Un esempio di questo tipo di attacco è rappresentato da un modello CycleGAN creato appositamente per non presentare uno specifico pattern a scacchiera nel suo rumore residuo, tipico delle immagini ottenute da modelli generativi[70] (Figura 2.5).

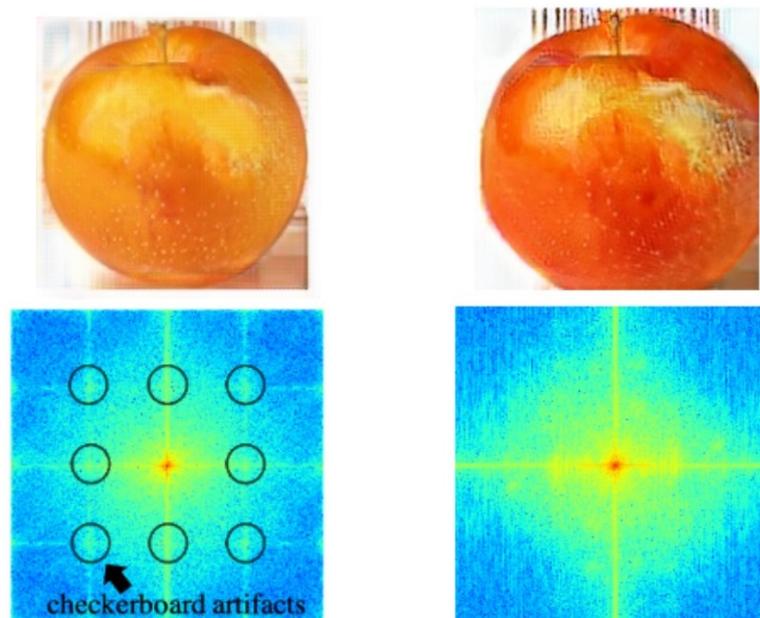


Figura 2.5: Confronto tra la noise fingerprint di un'immagine generata da una GAN tradizionale, in cui sono evidenti artefatti a scacchiera nel dominio delle frequenze, e quella di un'immagine prodotta da un modello CycleGAN, sviluppato per eliminare tali artefatti e, di conseguenza, modificare l'impronta residua del rumore.

## 2.2 Modelli per il riconoscimento di immagini sintetiche

### 2.2.1 Problemi nella classificazione delle immagini generate

Data la qualità visiva raggiunta dagli ultimi modelli di generazione, risulta evidente la necessità di tecniche e modelli in grado di classificare con alta precisione le immagini e il loro contenuto, distinguendole tra immagini reali e generate. A seguito di esperimenti applicati su un dataset contenente immagini reali e provenienti dai principali generatori, rappresentanti lo stato dell'arte, è stato dimostrato che con una CNN

classica (ResNet-50[36]) si è in grado di distinguere con un alto tasso di successo le immagini sintetiche ottenute da vari generatori[27].

Dai risultati dell'addestramento su ResNet-50 emerge che modelli di generazione con architetture simili o derivate sono in grado di influenzarsi reciprocamente nel processo di classificazione. Infatti, se vengono utilizzate immagini generate da un modello appartenente a una specifica categoria (GAN, Diffusion Model), il modello di classificazione non impara a riconoscere solo quelle ma migliora sensibilmente su tutti i modelli appartenenti alla categoria. Il problema emerge nel momento in cui è necessario classificare immagini di generatori con architetture differenti da quelli di provenienza delle immagini utilizzate per l'addestramento. Il classificatore, infatti, fallisce nel generalizzare su questo tipo di immagini, non avendo visto durante l'addestramento immagini con artefatti o noise fingerprint simili.

Con i metodi analizzati finora, una delle principali problematiche nel riconoscere e distinguere immagini sintetiche prodotte da modelli generativi e immagini reali è la necessità di disporre di esempi delle immagini dei generatori per addestrare i classificatori a riconoscerne le caratteristiche. Tuttavia, queste caratteristiche possono essere facilmente alterate nel caso in cui le immagini vengano compresse o ridimensionate, causando un degrado delle performance dei classificatori. In generale, se a un classificatore vengono presentate immagini sintetiche di generatori mai osservati durante l'addestramento o modificate, è probabile che queste vengano erroneamente classificate come reali, piuttosto che come sintetiche. Questo errore di classificazione si verifica perché il classificatore apprende a distinguere le immagini in base agli artefatti tipici dei singoli generatori; quando tali artefatti vengono eliminati a seguito di modifiche all'immagine, il sistema tende a considerarle reali[66].

Di seguito vengono presentati alcuni modelli e le rispettive architetture, impiegati per il riconoscimento di immagini sintetiche o particolarmente adatti a tale ambito grazie alle loro caratteristiche e potenzialità.

### 2.2.2 CLIP

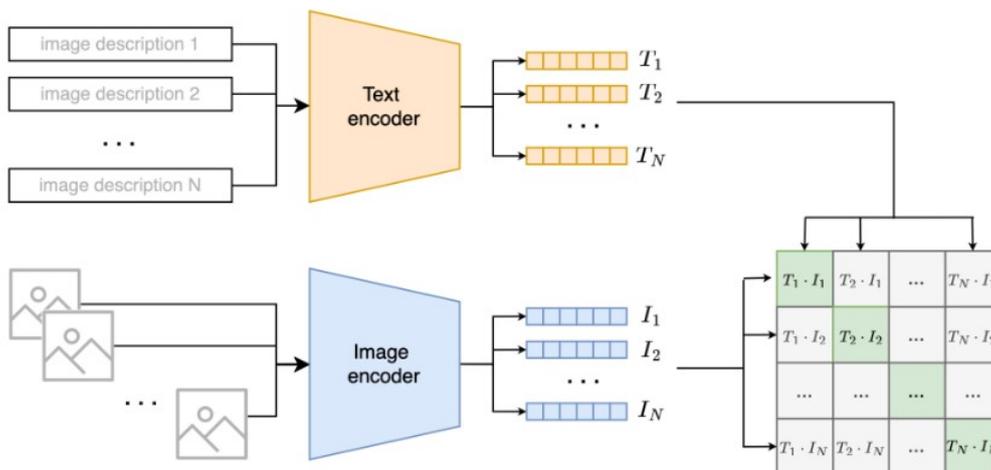


Figura 2.6: Architettura complessiva di CLIP.

Per affrontare questa limitazione, sono stati proposti nuovi classificatori basati su **Contrastive Language-Image Pre-training (CLIP)**[76], un vision-language model su

larga scala utilizzato per estrarre feature dalle immagini e da testo (Figura 2.6). L'idea consiste nell'utilizzare descrizioni (captions) di immagini reali per generare immagini sintetiche accoppiate a quelle reali e sfruttare una versione pre-addestrata di CLIP[76] per estrarre le feature corrispondenti alle immagini. Successivamente, un classificatore lineare viene addestrato utilizzando queste feature. I risultati ottenuti sono eccellenti sia su dataset "puliti" sia su dataset arricchiti mediante data augmentation, ovvero l'applicazione di trasformazioni (ad esempio, rotazioni, variazioni di luminosità, introduzione di rumore, ecc.). Ciò dimostra che il classificatore risultante è indipendente dagli artefatti specifici dei generatori[22].

Dato il successo di questo approccio, sono stati condotti ulteriori esperimenti per valutare le capacità dei modelli basati su CLIP[76], esplorando vari approcci per l'addestramento del classificatore in scenari con dataset limitati o immagini modificate[51]. Gli approcci analizzati includono: **Linear Probing**, **Fine-Tuning**, **Prompt Tuning** e **Adapter Network** (Figura 2.7).

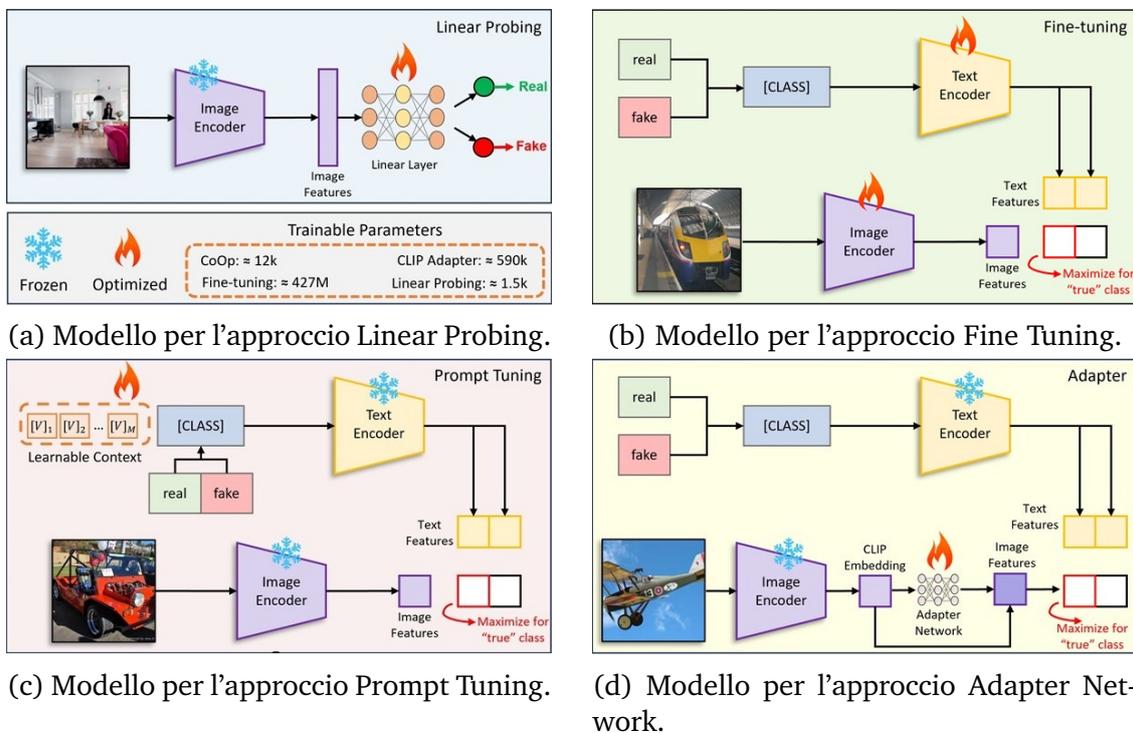


Figura 2.7: Confronto tra le architetture dei vari modelli di classificazione basati su CLIP.

Il metodo di **Linear Probing** (Figura 2.7a) si basa sull'aggiunta di un classificatore lineare in coda all'encoder visivo di CLIP congelando i parametri del modello, similmente a quanto sperimentato finora. In questo approccio, l'encoder testuale presente in CLIP viene completamente ignorato, utilizzando esclusivamente l'encoder visivo. Sebbene questa tecnica sia semplice e computazionalmente leggera, il fatto che ignori la componente testuale di CLIP rappresenta un limite significativo, privando il modello delle informazioni semantiche. Nonostante ciò, questo approccio offre un metodo efficace per sfruttare CLIP in contesti con risorse limitate.

L'approccio di **Fine-Tuning** (Figura 2.7b) prevede l'addestramento di tutto il modello CLIP (inclusi sia l'encoder visivo che testuale). Questo approccio è efficace, ma

è computazionalmente oneroso, richiedendo elevate risorse e tempi di addestramento. Inoltre, con l'aumentare della dimensione del modello, il fine-tuning può diventare instabile, rendendo necessario l'uso di un learning rate basso.

Il **Prompt Tuning** (Figura 2.7c) è una tecnica di adattamento in cui, per personalizzare il comportamento di CLIP a un compito specifico, vengono aggiunti all'input dei token testuali un certo numero di vettori "prompt" che sono parametrizzati e ottimizzabili. Questi vettori, concatenati ai token originali, fungono da contesto aggiuntivo e vengono appresi durante il training, mentre tutti gli altri parametri del modello rimangono fissi (congelati). In questo modo si evita il costoso processo di aggiornamento dell'intero modello, ottenendo un approccio computazionalmente più leggero rispetto al Fine-Tuning tradizionale.

La **Adapter Network** (Figura 2.7d) utilizza un livello aggiuntivo leggero e ottimizzabile, denominato **CLIP-Adapter**, che può essere aggiunto a entrambe le componenti di CLIP. Complessivamente il modello riesce a integrare le conoscenze pre-addestrate del modello con quelle apprese dall'adapter.

A seguito delle valutazioni effettuate su modelli basati su GAN, Diffusion e modelli commerciali, è emerso che gli approcci che considerano sia l'encoder visivo che testuale di CLIP hanno in generale performance migliori rispetto a modelli unicamente basati sull'encoder visivo come Linear Probing. In aggiunta è stata confermata l'efficacia di questi metodi anche in caso di dataset aumentati con blur gaussiano e compressione JPEG, o con dataset di training fortemente ridotti (ad esempio, 20.000 immagini).

### 2.2.3 DINOv2

**DINOv2** è un vision model che adotta un approccio di self-supervised learning, ovvero un metodo di apprendimento che non richiede etichette manuali per ogni immagine ma che è in grado di apprendere direttamente dalla struttura intrinseca dei dati. L'obiettivo è quello di estrarre rappresentazioni visive robuste e generalizzabili, capaci di supportare una vasta gamma di task nel riconoscimento delle immagini[69].

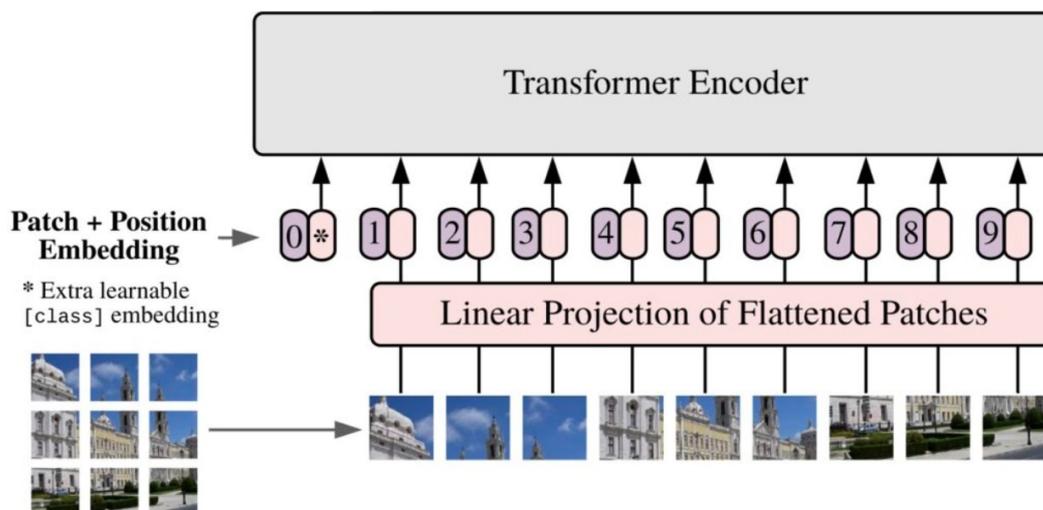


Figura 2.8: Schema concettuale del modello di un Vision Transformer.

Dal punto di vista architetturale, DINOv2 si fonda sull'architettura **Vision Transformer (ViT)**[26]. Il Vision Transformer suddivide l'immagine in patches, piccole sezioni

dell'immagine, e le trasforma in vettori, detti token, mediante una proiezione lineare per poi processarli con un transformer (Figura 2.8). Il cuore del modello è costituito da una serie di encoder basati sul meccanismo di self-attention, il quale consente di valutare l'importanza relativa di ciascun token rispetto agli altri, catturando così relazioni complesse tra le sezioni dell'immagine.

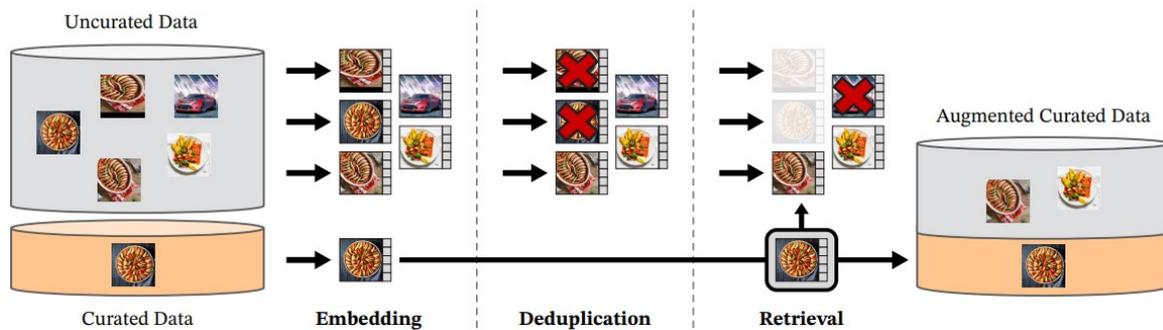


Figura 2.9: Pipeline per la creazione del dataset di training di DINOv2.

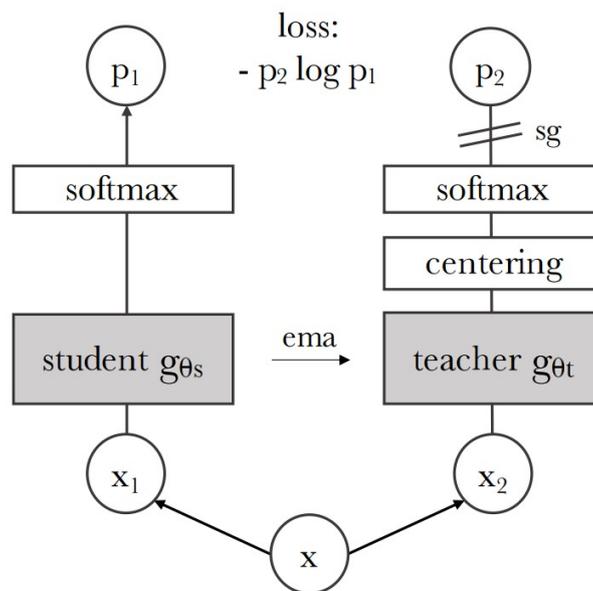


Figura 2.10: Schema del funzionamento di self distillation.

Una peculiarità di DINOv2 è la gestione dei dati per il training del modello. Questa si basa su un primo dataset di immagini selezionate la cui qualità è garantita e un secondo dataset di immagini raccolte da fonti differenti opportunamente filtrate. Questo secondo dataset viene processato da un'apposita pipeline (Figura 2.9) composta dalle seguenti fasi:

1. **Embedding:** ogni immagine di entrambi i dataset viene trasformata in un vettore di caratteristiche utilizzando un modello pre-addestrato. Questo processo permette di rappresentare le immagini in modo compatto facilitando i confronti tra immagini.

2. **Deduplication:** per garantire la diversità del dataset ed eliminare ridondanze, in questa fase sono identificate e rimosse le immagini duplicate o molto simili tra loro, sia all'interno delle immagini presenti nel secondo dataset che rispetto a quelle presenti nel dataset di immagini selezionate.
3. **Retrieval:** si selezionano le immagini del secondo dataset più simili (appartenenti alla stessa classe) alle immagini del primo. Questo processo permette di andare a comporre un dataset finale con un'ampia varietà dei dati di addestramento senza compromettere la qualità.

Il training del modello è stato realizzato tramite una tecnica chiamata **self distillation** (Figura 2.10) basato su un'architettura duale, composta da una rete teacher e una rete student. A entrambe le reti vengono presentate le stesse immagini sotto differenti trasformazioni realizzate tramite data augmentation oscurando alcune delle immagini per la rete student. Sia la rete student che la rete teacher, che non viene aggiornata tramite gradienti diretti ma attraverso una media mobile esponenziale (EMA) sui pesi della rete student, forniscono degli embedding per ogni immagine. L'obiettivo del training è quello di minimizzare la differenza tra gli embedding prodotti dalla rete student e quelli forniti dalla rete teacher, indipendentemente dalle variazioni introdotte dalle trasformazioni. Questo processo costringe il modello a imparare feature robuste e invarianti, capaci di catturare informazioni semantiche profonde senza l'uso di etichette manuali. Inoltre, grazie a questo meccanismo, DINOv2 può essere pre-addestrato su enormi dataset non etichettati, migliorando la sua capacità di generalizzazione su vari compiti di visione come classificazione e segmentazione[13].

## 2.2.4 Janus

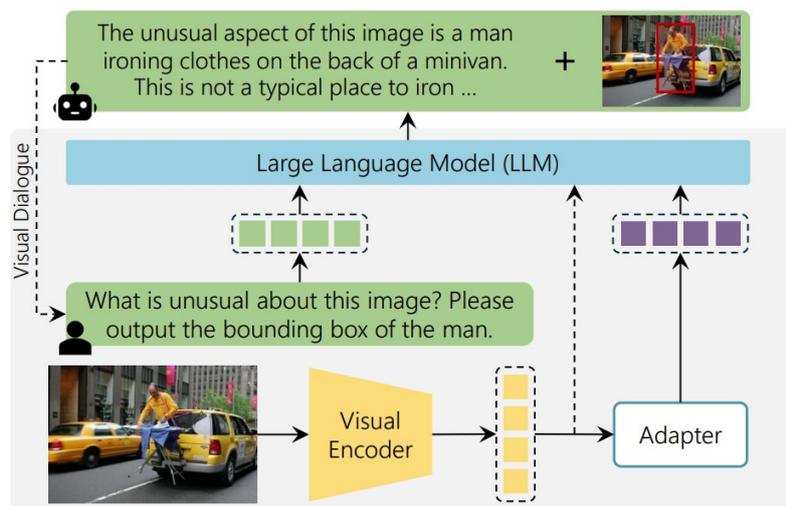
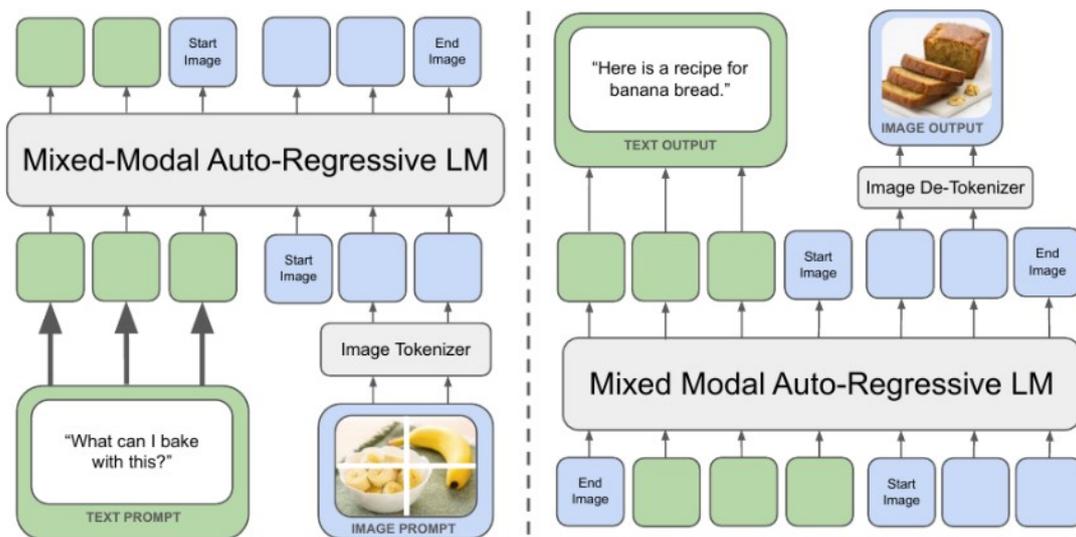


Figura 2.11: Architettura generale di un MLLM, composto da un visual encoder, un LLM e un adapter per trasporre le informazioni visive in formato testuale.

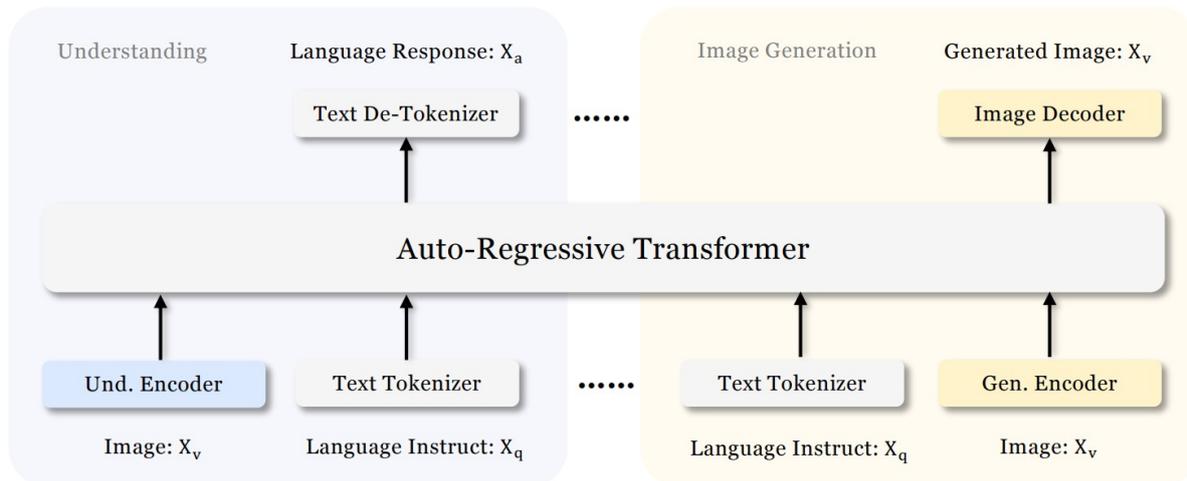
Janus[99] è un **Multimodal large language model (MLLM)**, ovvero un modello in grado di utilizzare informazioni visive e testuali in combinazione a un **Large language**

**model (LLM)** utilizzato per processare e comprendere il linguaggio naturale. Tipicamente, questo tipo di modelli genera output testuali in risposta a prompt che combinano testo e immagini; tuttavia, è possibile adattarli per la generazione di immagini mediante l'integrazione di componenti specializzate (Figura 2.11).

L'architettura di Janus si differenzia dalla struttura classica dei MLLM con una struttura duale che separa la parte di comprensione da quella di generazione. Tipicamente, infatti, i modelli MLLM utilizzano un unico image encoder sia per la parte di interpretazione e integrazione degli input testuali e visivi; Janus utilizza invece due image encoder differenti che si appoggiano sullo stesso LLM (Figura 2.12). Questa distinzione permette una maggiore specializzazione, garantendo output di qualità superiore in entrambi gli ambiti.



(a) Architettura di Chamaleon[92] con un singolo image encoder.



(b) Architettura duale di Janus con due encoder distinti.

Figura 2.12: Confronto tra le architetture di un MLLM tradizionale e Janus.

Oltre a Janus sono stati rilasciati due ulteriori modelli che ne rappresentano un'evoluzione e che rispondono a specifiche esigenze di efficienza, scalabilità e specializzazione nell'ambito della multimodal analysis. Il modello base, Janus, rappresenta l'approccio iniziale di integrazione tra input testuali e visivi, con un focus principale sulla

generazione di output testuali. JanusFlow[59] introduce, invece, una pipeline ottimizzata per il pre-processing dei dati riducendo le latenze e migliorando la gestione computazionale degli input aumentando la flessibilità del modello. Infine, JanusPro[17] mantiene l'architettura originale di Janus e perfeziona la procedura di addestramento del modello.

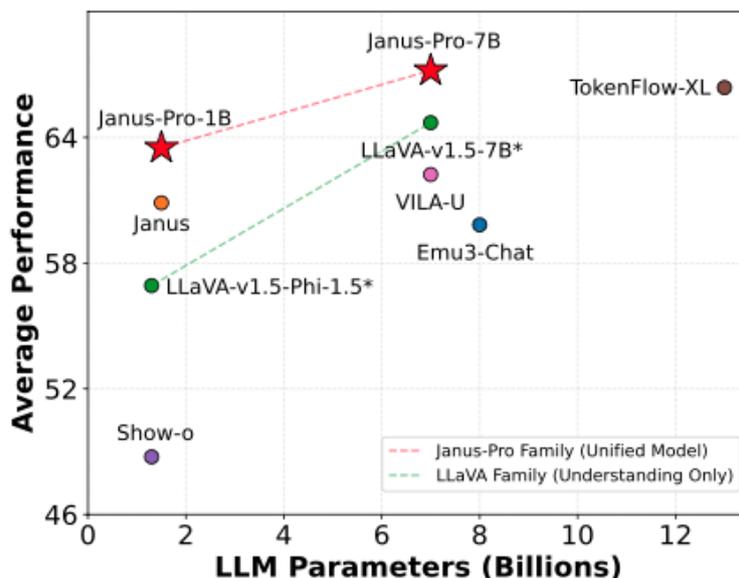


Figura 2.13: Accuracy media di JanusPro a confronto con gli altri principali MLLM calcolata su una serie di benchmark: POPE[57], MME-Perception[31], GQA[42] e MMMU[103].

JanusPro rappresenta l'attuale stato dell'arte nell'ambito del multimodal understanding (Figura 2.13), con risultati eccellenti anche in casistiche complesse che richiedono una profonda comprensione del contesto e la capacità di produrre output sia testuali che visivi di elevata qualità (Figura 2.14).

Image Description



This image depicts a **stunning coastal landscape** featuring a series of **tall, rugged rock** formations emerging from the ocean. The rocks are layered with sedimentary rock, giving them a distinct stratified appearance. The largest formation is prominently positioned in the foreground, with smaller ones receding into the distance. The ocean waves crash against the base of these formations, creating **white foam** and a **dynamic sense of movement**.

The **coastline is sandy**, with a narrow strip of beach visible on the right side of the image. The beach is bordered by steep cliffs, which rise dramatically from the shoreline. The cliffs are covered in patches of green vegetation, adding a touch of color to the otherwise earthy tones of the rocks.

The sky above is **overcast**, with thick clouds that suggest an impending storm or recent rainfall. The light is diffused, casting a soft, muted glow over the entire scene. Despite the cloudy sky, there is a sense of tranquility and natural beauty in this coastal landscape.

Describe this scene in detail.

General Knowledge



This cake is themed around the classic cartoon character **Tom and Jerry**. The cake features the characters Tom (the cat) and Jerry (the mouse) in a playful setting, with a backdrop that includes the words "Tom & Jerry". The cake itself is decorated with fondant shaped like cheese, which is a recurring theme in the cartoon series. The design also includes a small fondant house with a door, adding to the whimsical and playful theme of the cake.

Can you introduce the background story of this cake?

Text Recognition



The large black board has the following text:

**"Serving Soul since Twenty Twelve"**

Additionally, there are smaller sections with more detailed information, but the main heading is clearly visible.

What's written on the large black board?

Figura 2.14: Dimostrazione delle capacità di multimodal understanding di JanusPro.

# Capitolo 3

## Benchmark per la valutazione di modelli di classificazione

### 3.1 Problemi e limitazioni della valutazione dei modelli di classificazione

Negli ultimi anni, il numero di modelli generativi capaci di produrre immagini di qualità così elevata da risultare, a prima vista, indistinguibili da immagini reali è cresciuto in modo esponenziale. Nuovi modelli, sia open-source che commerciali, vengono continuamente sviluppati, migliorando progressivamente la qualità delle immagini generate. Parallelamente, si stanno sviluppando modelli in grado di distinguere i contenuti reali da quelli sintetici. Tuttavia, il confronto tra questi modelli di classificazione si presenta spesso complesso e problematico a causa della difficoltà nel reperire il codice sorgente delle soluzioni proposte e i dataset utilizzati nella loro valutazione.

Una delle principali difficoltà deriva dall'utilizzo di dataset limitati i quali, generalmente, includono immagini ottenute da un numero ristretto di modelli generativi. In altri casi, la valutazione viene effettuata utilizzando dataset creati ad hoc che, sebbene talvolta siano resi pubblici, non vengono adottati su larga scala. Questa frammentazione nella disponibilità dei dataset compromette la possibilità di stabilire un criterio di confronto unificato e oggettivo tra le diverse tecniche di classificazione.

Le problematiche descritte vengono ulteriormente accentuate nel caso in cui i dataset utilizzati comprendano unicamente immagini create con modelli generativi con architetture simili tra loro (es. modelli basati su GAN o Diffusion Model). Questa limitazione riduce la capacità di generalizzazione dei classificatori, rendendo difficile stimare le loro prestazioni in scenari più ampi e diversificati. Di conseguenza, il livello di efficacia effettivo di un classificatore potrebbe risultare sovrastimato rispetto alle condizioni reali di utilizzo.

Per superare queste limitazioni, è necessario disporre di dataset pubblici che includano immagini create con una vasta selezione di modelli generativi e che siano periodicamente aggiornati con immagini create con le nuove architetture emergenti. Inoltre, è opportuno sviluppare un **sistema di benchmarking** che consenta di valutare e confrontare i modelli di classificazione in modo sistematico e riproducibile, fornendo una piattaforma standardizzata per la valutazione delle loro prestazioni.

## 3.2 Soluzione proposta

Sulla base delle problematiche precedentemente descritte, emerge la necessità di un sistema che consenta di valutare in modo rigoroso e standardizzato i modelli di classificazione di immagini reali e sintetiche. Tale sistema deve basarsi su dataset comuni e predefiniti, composti da un'ampia varietà di immagini provenienti da diversi modelli generativi attualmente disponibili e da immagini reali.

Per rispondere a questa esigenza, in questo lavoro di tesi è stato sviluppato un benchmark per la valutazione di sistemi in grado di valutare le capacità dei modelli di classificazione. Il benchmark si articola su due componenti principali:

- **Un database di riferimento**, contenente immagini reali e immagini sintetiche generate da un vasto numero di modelli. Questo database è stato costruito prevalentemente a partire da dataset pubblici preesistenti, sia per le immagini reali che per quelle sintetiche.
- **Un framework di valutazione**, che consente di addestrare e valutare i modelli di classificazione sul database di riferimento, fornendo un'analisi dettagliata delle loro prestazioni e permettendo di confrontarli in maniera equa.

## 3.3 Caratteristiche del benchmark

### 3.3.1 Caratteristiche funzionali

I requisiti funzionali sono:

- **Gestione del database:** Per ogni immagine sintetica, devono essere conservate informazioni dettagliate sul modello generativo utilizzato, incluse eventuali tecniche di condizionamento applicate (ad esempio rumore, prompt testuale o immagine di riferimento). Se disponibili, i parametri specifici del condizionamento e le informazioni sul contenuto delle singole immagini devono essere registrati nel database.
- **Data augmentation:** il benchmark supporta diverse tecniche di data augmentation, ovvero trasformazioni di varia tipologia da applicare sulle immagini del database per aumentarne l'eterogeneità. Tale necessità emerge dal fatto che alcuni modelli, come CLIP[76], utilizzati in letteratura per il riconoscimento di immagini reali e sintetiche, prevedono una fase di pre-processing con trasformazioni finalizzate ad aumentare la qualità e la robustezza delle feature apprese.
- **Valutazione dei modelli:** il benchmark fornisce metriche standard per analizzare le prestazioni dei classificatori valutati, come accuracy, precision, recall e F1-score. Le metriche si occupano di valutare tre aspetti distinti:
  - La capacità del modello di distinguere tra immagini reali e sintetiche.
  - L'accuratezza nell'attribuire un'immagine sintetica al modello generativo che l'ha prodotta.
  - La valutazione della capacità dei singoli modelli di riconoscere immagini create con generatori non presenti nel database utilizzato per l'addestramento.

Le metriche sono accessibili tramite un'interfaccia grafica intuitiva che consente la visualizzazione dettagliata dei risultati.

- **Flessibilità nell'addestramento:** il benchmark supporta diverse modalità di addestramento come training completo su tutto il database e approcci di continual learning in modo da offrire diverse panoramiche su come i modelli utilizzati si comportano in differenti scenari.
- **Strategia di aggiornamento del database:** il database è sviluppato per supportare aggiornamenti periodici con l'inserimento di immagini create con nuovi modelli generativi man mano che vengono rilasciati. Il benchmark fornisce un meccanismo chiaro per integrare nuovi dataset senza compromettere la coerenza e la riproducibilità delle valutazioni precedenti.

### 3.3.2 Considerazioni sulla qualità del database

La qualità del database del benchmark è essenziale per ottenere valutazioni affidabili e significative dei modelli di classificazione. Per garantire una corretta generalizzazione ed evitare bias, il database rispetta specifiche caratteristiche.

Gli aspetti fondamentali su cui è basata la costruzione del database sono la diversità delle immagini e la copertura dei modelli generativi. Il database include immagini prodotte da un'ampia gamma di generatori, coprendo diverse architetture e varianti di ciascun modello. Questo permette di valutare la capacità di generalizzazione dei classificatori e di evitare che essi si adattino unicamente a specifici pattern caratteristici di alcune famiglie di generatori. Inoltre, viene mantenuta un'ampia varietà visiva nelle immagini selezionate, includendo variazioni di risoluzione, contenuti, illuminazione e composizione, così da evitare che il classificatore apprenda caratteristiche irrilevanti per la classificazione invece di distinguere in modo robusto le immagini sintetiche dalle reali.

Un altro elemento chiave è il bilanciamento tra immagini reali e immagini sintetiche. Un database squilibrato potrebbe portare a modelli di classificazione che tendono a favorire la classe predominante, riducendone l'efficacia in scenari reali. Viene quindi garantita una distribuzione bilanciata in termini di quantità e categorie semantiche delle immagini, evitando che il classificatore si specializzi eccessivamente su un sottoinsieme specifico. In particolare è fondamentale che le immagini generate da ciascun modello coprano una gamma diversificata di soggetti. Se, ad esempio, per un generatore fossero incluse nel database esclusivamente immagini di volti e per un altro solo immagini di edifici, si introdurrebbe un bias che comprometterebbe la capacità del classificatore di distinguere tra immagini reali e sintetiche in modo indipendente dal contenuto. Per mitigare tale problema, il database include, quando possibile, una varietà di categorie all'interno di ogni gruppo di immagini provenienti dallo stesso generatore, assicurando che la distribuzione sia il più variegata e bilanciata possibile.

Infine, per simulare scenari d'uso realistici, viene considerata anche l'eventualità che le immagini subiscano alterazioni o degradazioni, come compressioni o filtri. Per simulare scenari d'uso reali, è utile includere una selezione di immagini modificate tramite:

- Compressione lossy (es. JPEG con diversi livelli di qualità);
- Applicazione di filtri (blur, sharpening, distorsioni);

- Ridimensionamento con perdita di dettagli.

L'inclusione di tali trasformazioni permette di valutare la robustezza dei modelli di classificazione rispetto a condizioni non ideali, fornendo una misura più realistica delle loro capacità nel distinguere immagini reali da sintetiche.

### 3.3.3 Modalità di addestramento previste

Come già descritto in precedenza, il benchmark supporta diverse strategie di addestramento per valutare l'efficacia dei modelli di classificazione in scenari differenti. Ogni strategia ha implicazioni specifiche sulla capacità di generalizzazione del modello e sulla gestione dei dati di training.

- **Full Training:** rappresenta la strategia di addestramento tradizionale, in cui il modello viene allenato su tutto il database disponibile in un'unica fase. Questo metodo consente di valutare la capacità del modello di distinguere tra immagini reali e sintetiche nel suo complesso, ma non permette di osservare il suo adattamento progressivo a nuovi generatori.
- **Sliding Window:** con questa strategia, l'addestramento avviene in più fasi, partendo da un subset iniziale del database contenente immagini generate dai primi modelli generativi rilasciati in ordine cronologico. Ad ogni nuova fase, vengono aggiunti nuovi generatori al dataset di training e si addestra un nuovo modello da zero. Ogni modello viene valutato sia sui generatori inclusi nell'addestramento, sia su quelli ancora non visti, permettendo di osservare come cambia la generalizzazione al variare della quantità e qualità delle immagini sintetiche nel tempo. Tuttavia, questo approccio comporta un elevato costo computazionale e temporale, poiché richiede di addestrare molteplici modelli, ciascuno con un numero crescente di dati di training.
- **Continual Learning:** simile al metodo Sliding Window, con la differenza che lo stesso modello viene aggiornato progressivamente invece di riaddestrarne uno nuovo ogni volta. L'addestramento inizia con un subset iniziale di generatori, ai quali vengono aggiunti progressivamente quelli successivi. Anche in questo caso, si può valutare la capacità di generalizzazione del modello su generatori non ancora visti ma, in aggiunta, si può anche osservare se il modello mantiene la capacità di classificare correttamente le immagini appartenenti ai generatori appresi nelle fasi precedenti. Dal punto di vista computazionale, questo approccio è più efficiente dello Sliding Window, poiché viene utilizzato un unico modello che viene aggiornato nel tempo invece di dover addestrare modelli separati ad ogni fase. Tuttavia, un potenziale problema potrebbe essere il **catastrophic forgetting**, ovvero la perdita di conoscenza su generatori più vecchi man mano che il modello apprende nuove informazioni.

### 3.3.4 Caratteristiche tecniche

Le caratteristiche tecniche necessarie per il corretto funzionamento del benchmark sono:

- **Interoperabilità:** il database, utilizzato sia per la composizione che per la valutazione dei modelli, segue uno standard predefinito, permettendo una completa compatibilità tra i diversi componenti del benchmark.
- **Modularità:** la gestione del database è modulare, in modo che ogni dataset scaricato sia indipendente dagli altri. Ciò consente di rimuovere facilmente dataset obsoleti o di integrarne di nuovi per ampliare la copertura dei modelli generativi considerati.
- **Riproducibilità e determinismo:** il database deve essere ricavabile a partire dai dataset pubblici utilizzati per la sua creazione. Per garantire la confrontabilità dei risultati, il benchmark assicura che ogni generazione del database e ogni fase di data augmentation siano deterministiche. Questo significa che, indipendentemente dalle esecuzioni, lo stesso input deve sempre produrre il medesimo database finale.
- **Scalabilità ed efficienza computazionale:** il benchmark gestisce dataset di grandi dimensioni e supporta l'addestramento di modelli su infrastrutture hardware differenti, assicurando un utilizzo ottimale delle risorse disponibili.

## 3.4 Architettura del benchmark

Il benchmark è organizzato in due moduli principali: la gestione del database e l'addestramento e valutazione dei modelli di classificazione. Il primo modulo riguarda il download dei dataset pubblici, la selezione delle immagini e la loro standardizzazione in un formato unificato. Il secondo modulo gestisce l'applicazione delle pipeline di data augmentation, la configurazione e l'addestramento dei modelli di classificazione, oltre alla generazione delle metriche di valutazione.

### 3.4.1 Database

#### Processo di costruzione complessivo

Il processo di creazione del database (Figura 3.1) è articolato nelle seguenti fasi:

1. **Download dataset:** i dataset necessari vengono scaricati. A seconda del dataset, può essere disponibile uno script per l'automatizzazione del processo.
2. **Selezione delle immagini per il dataset sintetico:** per ciascun dataset di immagini sintetiche, vengono selezionate le immagini che faranno parte del dataset finale.
3. **Creazione dei dataset sintetici parziali:** le immagini selezionate vengono organizzate in dataset intermedi. Se sono presenti più dataset di immagini sintetiche, il processo di selezione e creazione viene ripetuto per ciascuno di essi.
4. **Aggregazione del dataset sintetico:** i dataset parziali vengono combinati in un unico dataset contenente tutte le immagini sintetiche selezionate.

5. **Selezione delle immagini associate dai dataset reali:** utilizzando il campo *Paired real images* del dataset sintetico, vengono identificate e selezionate le immagini reali corrispondenti.
6. **Selezione delle immagini rimanenti dai dataset reali:** per bilanciare il dataset, vengono selezionate ulteriori immagini reali fino a raggiungere il numero richiesto, creando così il dataset finale di immagini reali.
7. **Unione dei dataset reali e sintetici:** i due dataset vengono combinati in un unico database bilanciato, che costituisce la base per le fasi di addestramento e valutazione.
8. **Train/Validation split:** il database viene suddiviso in set di addestramento e validazione, secondo una proporzione predefinita.
9. **Applicazione pipeline base di data augmentation:** prima dell'addestramento di un modello, sul training set e sul validation set vengono applicate le rispettive pipeline di data augmentation per migliorare la generalizzazione del modello.
10. **Applicazione post-processing:** se il modello da addestrare richiede l'applicazione di trasformazioni di post-processing specifiche, sia per quanto riguarda il training set che per il validation set, queste vengono applicate.

### Creazione del database

Il database viene creato a partire da immagini provenienti da diverse fonti, che vengono unificate, filtrate e standardizzate secondo un modello comune. Il database è progettato per poter archiviare, per ogni immagine in esso contenuta, le seguenti informazioni:

- **Label:** indica se l'immagine è reale o sintetica.
- **Generator:** specifica il modello utilizzato per la generazione dell'immagine.
- **Filename:** percorso del file all'interno del dispositivo, riferito all'immagine originale.
- **Description:** classe semantica dell'immagine, ossia la categoria o il soggetto rappresentato.
- **Positive prompt:** testo utilizzato per condizionare la generazione dell'immagine, tipicamente una descrizione dettagliata delle caratteristiche desiderate.
- **Negative prompt:** testo che definisce gli elementi da escludere nella generazione dell'immagine.
- **Conditioning:** categoria di input iniziale fornito al modello generativo, come testo, immagini di riferimento o rumore.
- **Origin dataset:** dataset originale da cui proviene l'immagine.

- **Paired real images:** elenco di immagini reali associate all'immagine sintetica, ovvero immagini che hanno influenzato la sua generazione, presenti negli altri dataset considerati.
- **Found in datasets:** elenco di altri dataset considerati in cui l'immagine è presente, utile per evitare duplicazioni nel dataset finale.

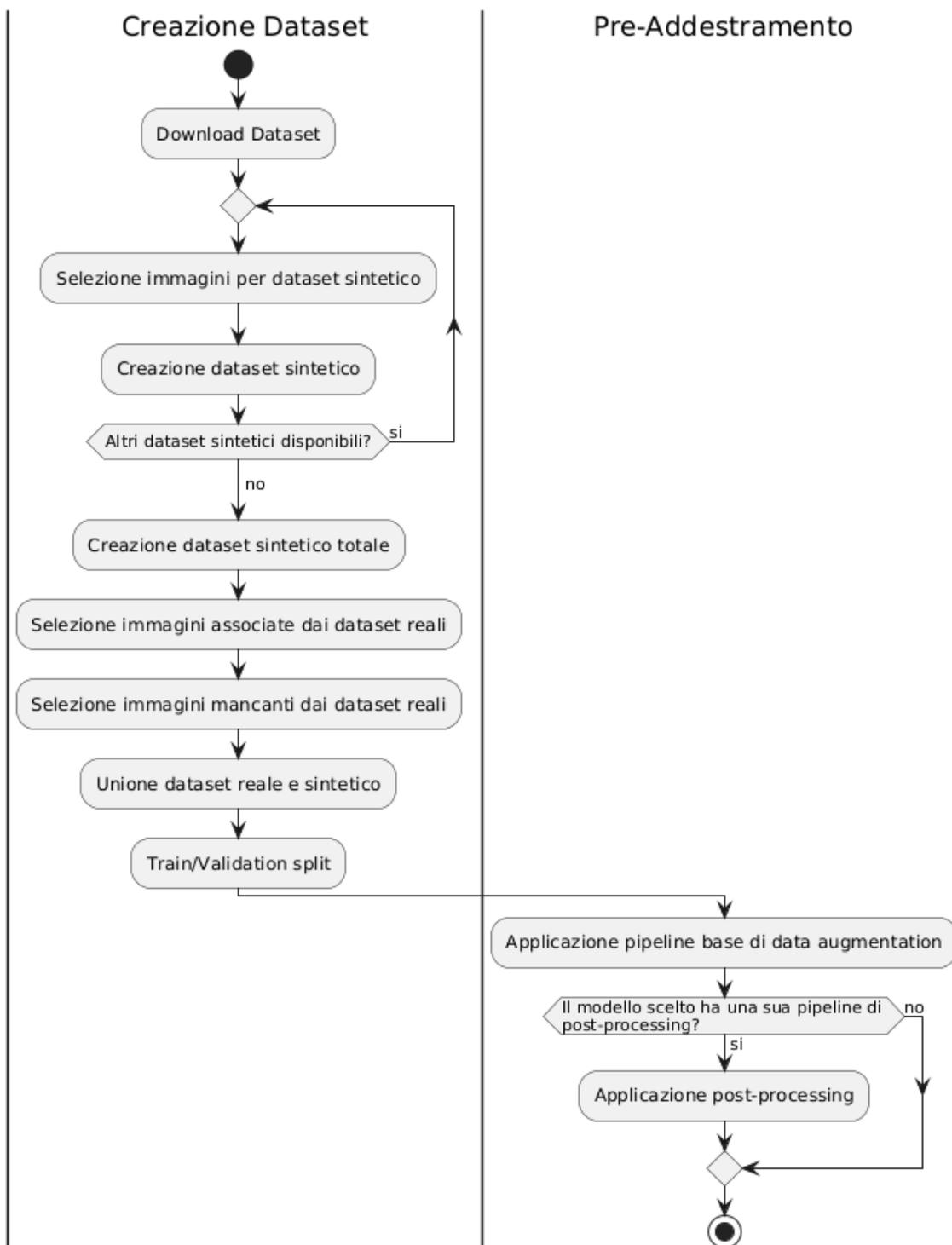


Figura 3.1: Processo di creazione del database.

Le informazioni vengono estratte direttamente dalle fonti quando disponibili. I campi obbligatori sono *Label*, *Generator*, *Filename* e *Origin dataset* mentre i restanti sono inseriti qualora le informazioni siano presenti nel dataset.

Alcuni metadati, come *Conditioning Positive prompt*, *Negative prompt* e *Paired real images*, si applicano esclusivamente alle immagini sintetiche e sono quindi omessi per le immagini reali. Altri campi, come *Description* sono invece presenti per entrambe le tipologie di dataset.

Considerando la necessità di integrare immagini provenienti da diverse fonti, ciascuna con standard e strutture eterogenee, è essenziale adottare un'architettura modulare che consenta l'integrazione di questi dataset senza doverli gestire singolarmente.

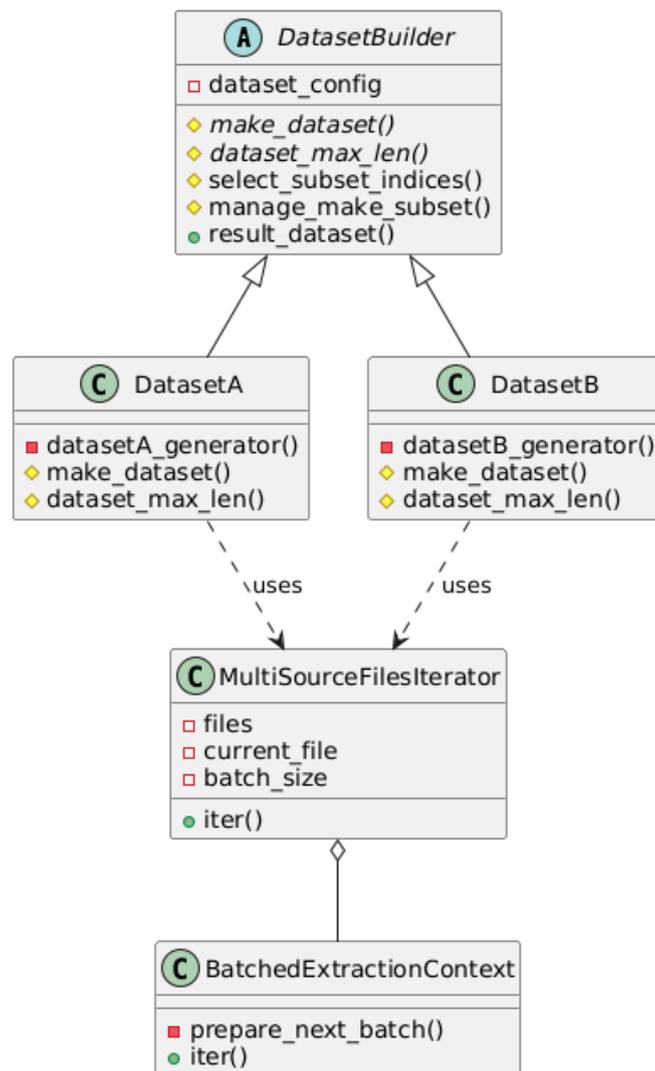


Figura 3.2: Architettura per la gestione dei dataset di immagini sintetiche.

L'architettura (Figura 3.2) consente un'integrazione scalabile dei dataset di immagini sintetiche. Per aggiungere un nuovo dataset, è sufficiente estendere una specifica classe, adattandola alla struttura e alla composizione del dataset da includere. Nello schema UML, questa strategia di gestione dei dataset è rappresentata dalle classi DatasetA e DatasetB, delle classi fittizie rappresentanti due ipotetici dataset usati dal benchmark, ognuna delle quali gestisce la struttura a livello di file system del rispettivo

dataset e fornisce alla classe `DatasetBuilder` le informazioni necessarie per selezionare la porzione di dati da utilizzare.

Ogni classe dedicata a un dataset costruisce una sezione del dataset finale in modo indipendente, trattandolo come un'entità autonoma.

L'integrazione dei dati avviene attraverso le classi `MultiSourceFilesIterator` e `BatchedExtractionContext`, le quali consentono:

- La gestione delle immagini in batch, ottimizzando il processo di caricamento e pre-elaborazione.
- Il supporto per dataset costituiti da archivi compressi, evitando la necessità di lavorare esclusivamente con file singoli.

Le diverse sezioni del dataset vengono successivamente unite per costituire il dataset sintetico complessivo.

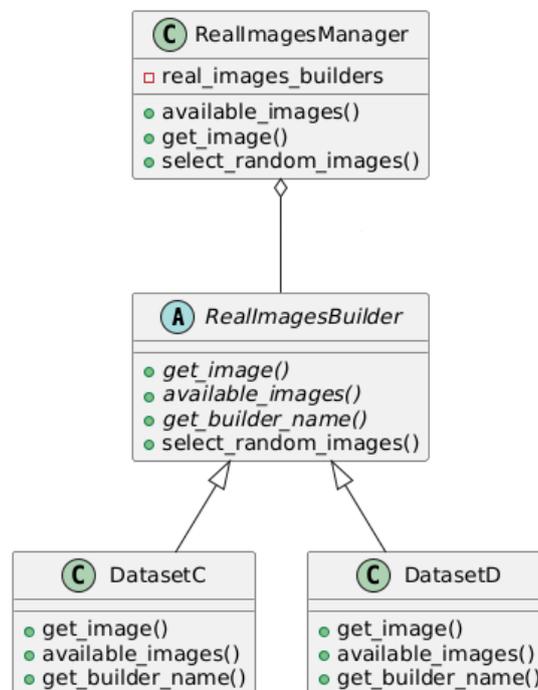


Figura 3.3: Architettura per la gestione dei dataset di immagini reali.

La gestione dei dataset di immagini reali avviene in modo simile (Figura 3.3), la classe `RealImagesManager` si occupa di gestire collettivamente i singoli dataset, le cui caratteristiche di ognuno sono modellate tramite un'apposita classe che estende `RealImagesBuilder`.

La principale differenza tra la gestione delle immagini reali e quelle sintetiche risiede nel processo di integrazione: mentre per i dataset sintetici le immagini vengono inizialmente trasformate singolarmente e successivamente unite, nel caso dei dataset reali le immagini vengono selezionate da tutti i dataset disponibili e strutturate direttamente in un unico dataset di immagini reali.

Questa differenza è dovuta all'incertezza riguardo al numero di immagini da selezionare per ciascun dataset reale, il che comporta la necessità di una strategia collettiva per decidere quali immagini includere. Di conseguenza, il processo di selezione delle

immagini reali richiede una gestione più centralizzata per garantire una composizione equilibrata e coerente dell'intero dataset di immagini reali.

### Standardizzazione del database

Per garantire uno standard coerente nella gestione dei dataset, è stata adottata la libreria **Datasets** di **Hugging Face**[54]. Questa libreria è altamente scalabile e ottimizzata per la gestione di grandi volumi di dati, offrendo un'API uniforme che ne semplifica operazioni comuni come caricamento, trasformazione, filtraggio e campionamento.

Un aspetto chiave della libreria **Datasets** è la sua integrazione nativa con i principali framework di machine learning, tra cui **PyTorch**[72] e **TensorFlow**[1]. Questa compatibilità consente di convertire facilmente i dataset in tensori ottimizzati per l'addestramento dei modelli, migliorando l'efficienza computazionale.

Per l'archiviazione dei dati, la libreria utilizza il formato **Apache Arrow**[30], che garantisce un accesso rapido ed efficiente ai dati, ottimizzando le operazioni di I/O. Inoltre, ogni dataset mantiene una cartella di cache locale, utilizzata durante la creazione del database, per mantenere i dataset parziali reali e sintetici. Questo approccio riduce la necessità di rielaborazioni ridondanti e migliora significativamente le prestazioni complessive.

### Selezione delle immagini

Il database è stato costruito selezionando 5000 immagini sintetiche per ciascun generatore incluso (Tabelle 3.1 e 3.2), bilanciandole con un numero equivalente di immagini reali. Questo bilanciamento tra classi reali e sintetiche è fondamentale per evitare che, durante la fase di addestramento, il modello apprenda a prediligere una classe rispetto all'altra.

Per assicurare una copertura ampia e diversificata delle immagini sintetiche, sono stati individuati e selezionati diverse fonti contenenti immagini generate da un'ampia gamma di modelli generativi.

La selezione delle immagini sintetiche è stata effettuata considerando i seguenti criteri per ciascun dataset analizzato:

- **Presenza e distribuzione dei generatori:** identificazione dei modelli generativi presenti nelle fonti e quantificazione delle immagini disponibili per ciascuno di essi.
- **Diversità delle classi rappresentate:** valutazione del contenuto delle immagini per garantire, per ogni generatore, un set di immagini il più diversificato possibile in termini di soggetti e caratteristiche visive.
- **Qualità e risoluzione delle immagini:** valutazione della risoluzione e della qualità complessiva delle immagini per favorire una maggiore variabilità e robustezza nel database.

L'obiettivo di questi criteri è costruire un dataset eterogeneo anche tra le immagini sintetiche prodotte dallo stesso generatore, in modo da migliorare la generalizzazione del modello durante l'addestramento e, conseguentemente, la qualità del benchmark.

La selezione delle immagini reali è stata guidata dando priorità alle immagini associate, ovvero le immagini utilizzate per il condizionamento di una o più delle immagini



Tabella 3.2: Distribuzione della selezione delle immagini nei dataset selezionati per i generatori da Palette[84] (10/11/2021) a Flux.1 Schnell[53] (01/08/2024).

Nome generatore	Aeroblade[82]	Artifact[78]	Detecting GNN Generated [95]	DMImage Detection[21]	DRCT-2M[15]	ELSA_D3[2]	GemImage[106]	Imaginet[10]	PolarDiff-shield[6]	Selfshgene[9]	Synthbusters [7]	Towards the Detection of Diffusion Models[81]
Palette[84]		5000										
VQ-Diffusion[35]		2500					2500					
Denoising Diffusion GAN[100]		5000										
GLIDE[64]		1600		900			1500		500		500	
Latent Diffusion[83]		2000		1500								1500
MAT[56]		5000										
Diffusion GAN[96]		2500										2500
Stable Diffusion 1.4[20]					1000	3000	500				500	
Stable Diffusion 1.5[20]	500				2500		2000					
Stable Diffusion 2.1[20]	500				1000	2000		1000	500			
Wukong[34]							5000					
Midjourney 5[93]							5000					
DeepFloyd-IF[3]						5000						
Stable Diffusion XL 1.0[20]					1000	1500		500	500	1000	500	
DALL-E V3[68]								4000	500		500	
Stable Diffusion Turbo[20]												
Stable Diffusion XL Turbo[20]						5000						
Flux.1 dev[53]										5000		
Flux.1 schnell[53]										5000		

generate, identificate attraverso il campo *Paired real images* del database (se presente). Questa scelta è motivata dalla necessità di garantire la presenza di immagini correlate tra la parte reale e quella sintetica del database, spesso simili nel contenuto. Tale approccio favorisce l'addestramento di modelli che si basano sulle caratteristiche intrinseche delle immagini generate piuttosto che sui soggetti rappresentati, migliorando la fedeltà dei risultati in fase di valutazione. Una volta selezionate le immagini reali associate, il database viene completato integrando ulteriori immagini reali fino a raggiungere lo stesso numero di immagini sintetiche.

Inoltre, durante la fase di costruzione del dataset, è stato applicato un ulteriore filtro dimensionale per escludere tutte le immagini che non rientravano in un intervallo di dimensioni predefinito. Questo filtro è stato introdotto per:

- Eliminare immagini non scaricate correttamente, tipicamente registrate con dimensioni anomale (es.  $1 \times 1$  pixel).
- Escludere immagini con una risoluzione troppo bassa per essere utili ai fini dell'addestramento (non rappresentando un caso di prova realistico).
- Limitare l'inclusione di immagini eccessivamente grandi, che potrebbero rallentare significativamente le operazioni di pre-processing ed elaborazione.

Poiché questa selezione avviene dinamicamente durante la costruzione del database, è stato necessario prevedere un numero iniziale di immagini superiore a quello richiesto per ciascun dataset e generatore (Tabelle 3.1 e 3.2), in modo da garantire il numero minimo di immagini anche dopo il filtraggio.

### Gestione della data augmentation

Il database così composto include un totale di **390.000 immagini**, comprendente 39 generatori e provenienti da 16 fonti (12 per le immagini sintetiche e 4 per le reali), equamente suddivise tra le due classi (reale e sintetica). Per la parte destinata al training set, che costituisce l'**80%** dell'intero database, viene applicata una pipeline di *data augmentation* (Figura 3.4) strutturata nei seguenti passaggi:

1. **Ridimensionamento delle immagini:** Considerando l'elevato numero di immagini e la necessità di applicare trasformazioni successive, si è deciso di ottimizzare il carico computazionale e ridurre i tempi di data augmentation. Pertanto, tutte le immagini con dimensioni superiori a  $1080 \times 1080$  vengono ridimensionate, ritagliando una porzione pari a  $1080 \times 1080$ . Inoltre, per il 20% delle immagini totali viene eseguito un ulteriore ridimensionamento, che prevede l'estrazione di una porzione  $256 \times 256$  dall'immagine originale.
2. **Modifiche cromatiche:** per aumentare la diversità del database, sull'**80%** delle immagini viene applicata una trasformazione dei parametri cromatici, alterando contrasto, saturazione e tonalità in modo casuale.
3. **Conversione in scala di grigi:** con una probabilità del **20%**, le immagini vengono convertite in scala di grigi.
4. **Occlusioni, rumore e compressione:** per simulare condizioni di acquisizione reali e introdurre variazioni strutturali nelle immagini, vengono applicate le seguenti trasformazioni con differenti probabilità:

- *Cutout*: occlusioni di alcune porzioni delle immagini.
- *Rumore gaussiano*: aggiunta di perturbazioni casuali per aumentare la robustezza del modello.
- *Blur*: applica una sfocatura casuale all'immagine per simulare variazioni di messa a fuoco.
- *Compressione lossy*: riduzione controllata della qualità dell'immagine per emulare artefatti di compressione comuni nei formati JPEG e altri.

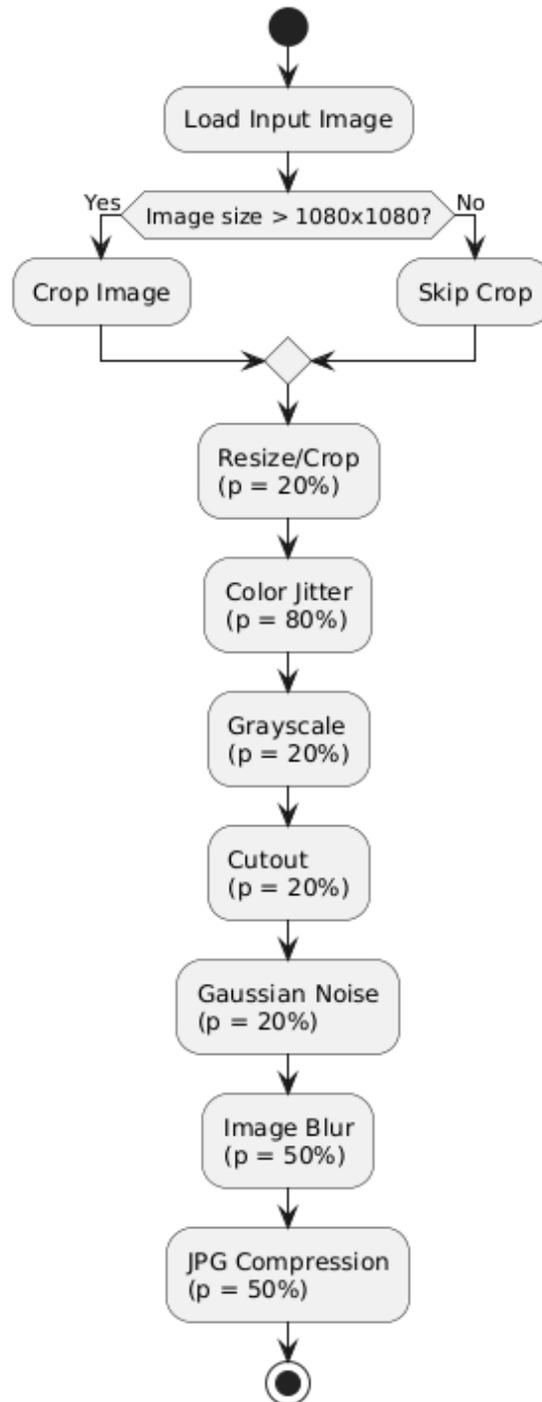


Figura 3.4: Diagramma della pipeline di data augmentation per il training set.

Questa pipeline viene applicata al database prima della fase di addestramento. Poiché ogni trasformazione è deterministica, il database risultante rimane invariato tra diverse esecuzioni, garantendo coerenza nella fase di training e comparabilità dei risultati ottenuti.

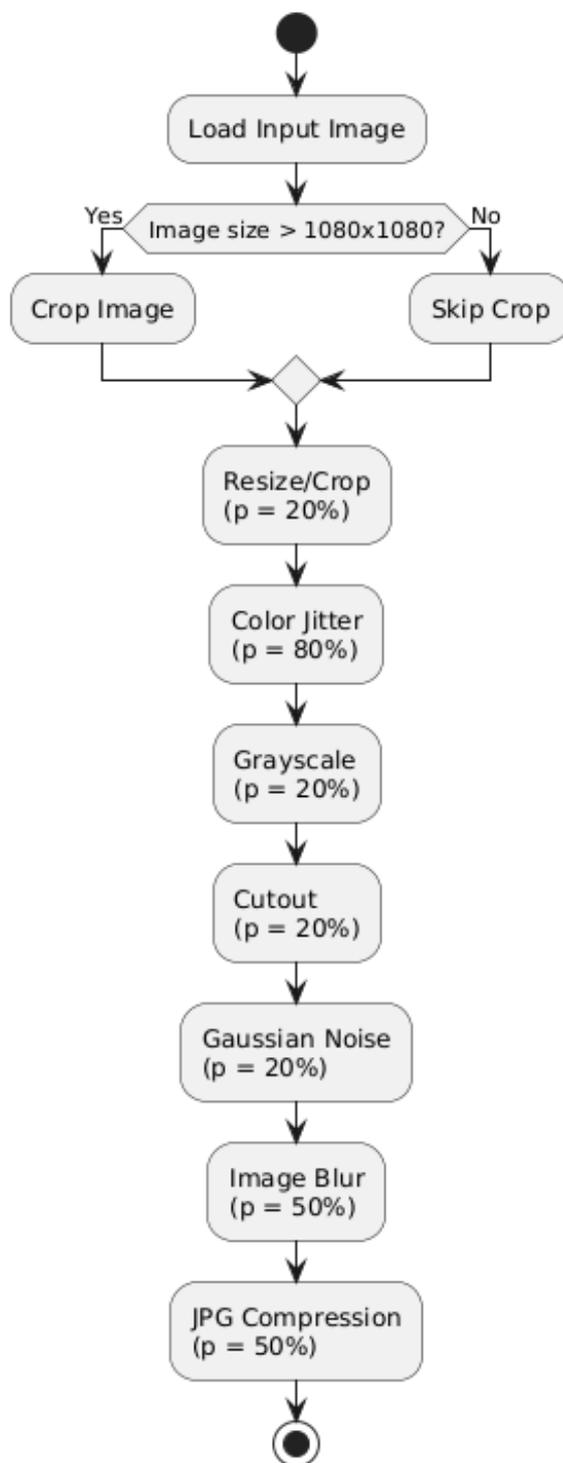


Figura 3.5: Diagramma della pipeline di data augmentation per i set di validation e test.

Analogamente è presente una specifica pipeline applicata sul set di validation e test

(Figura 3.5). Questa, oltre ad applicare trasformazioni simili a quelle applicate sul training set come rumore gaussiano e conversione in scala di grigi, applica tre compressioni, ognuna con probabilità decrescente (50%, 20%, 10%). L'iter di compressione ripetuta rappresenta una simulazione realistica delle degradazioni che le immagini subiscono durante il loro ciclo di vita digitale. In numerosi contesti, le immagini vengono scaricate, condivise e ripubblicate, venendo sottoposte a successivi processi di compressione, tipicamente tramite algoritmi lossy come JPEG. Ogni compressione comporta una perdita di informazione, generando artefatti che si accumulano progressivamente (Figura 3.6). L'applicazione di molteplici compressioni in ambiente controllato consente, quindi, di replicare tali degradazioni e di avvicinare le immagini a quelle effettivamente presenti sul web, migliorando così la robustezza e la generalizzazione in uno scenario altamente realistico.



Figura 3.6: Progressiva alterazione dell'immagine originale (a) dopo una (b), due (c) e tre (d) compressioni JPG.

Successivamente, in funzione delle necessità del modello di classificazione adottato, è possibile applicare una seconda fase di *post-processing*, sia sul *training set* che sul

*validation set* (Figura 3.7). Quest’ultima fase è specifica per l’architettura del modello utilizzato e può includere trasformazioni aggiuntive ottimizzate per il miglioramento delle prestazioni.

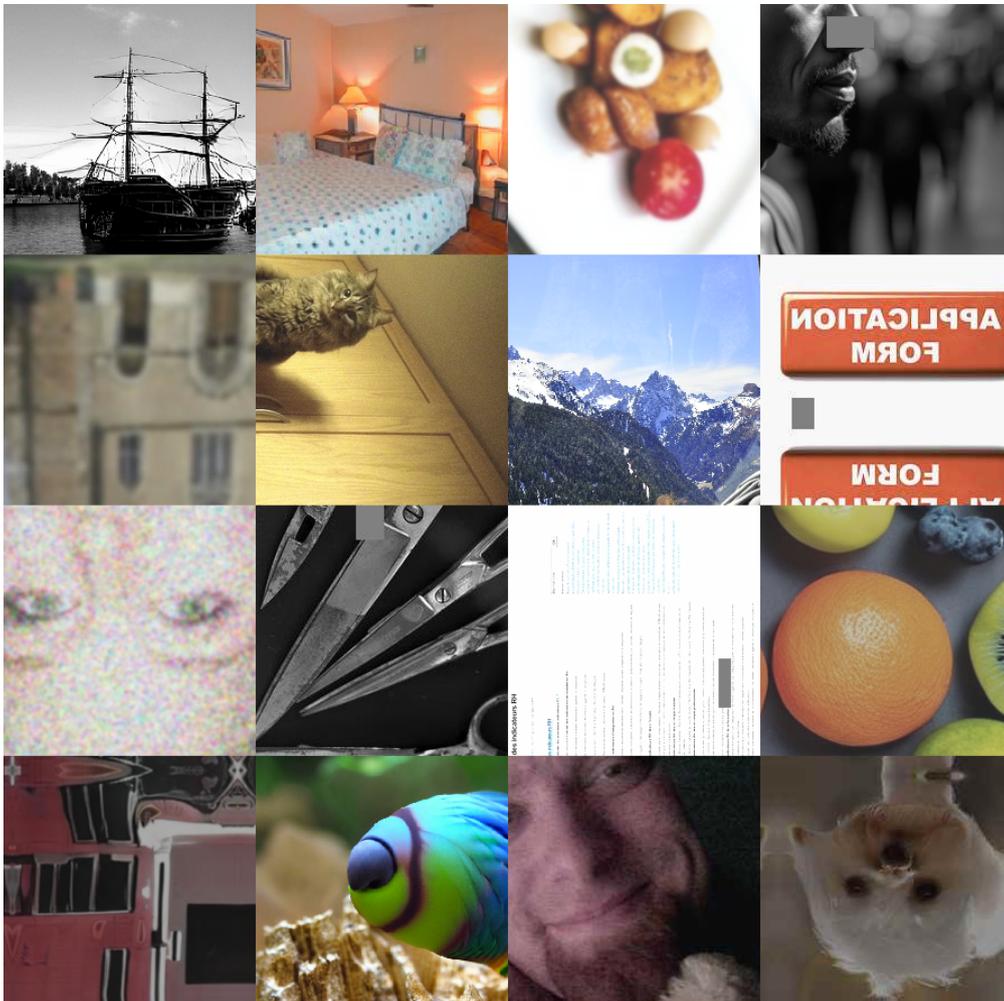


Figura 3.7: Esempi di immagini risultate dalla pipeline del training set in combinazione con il post-processing di CLIP[76].

### 3.4.2 Configurazione, addestramento e valutazione dei modelli

Il benchmark consente di configurare in modo flessibile l’addestramento dei classificatori integrati, permettendo la selezione del modello e della strategia di training. Il benchmark nel suo complesso si appoggia a **PyTorch Lightning**[29], un framework che garantisce una grande flessibilità nella gestione dei modelli e degli addestramenti. La configurazione avviene tramite un file dedicato, che definisce tutti i parametri rilevanti per l’addestramento, garantendo modularità e riproducibilità degli esperimenti.

I principali parametri configurabili includono:

- **Iperparametri di training:** numero massimo di epoche, batch size, dimensione di input del modello, learning rate e relativo scheduler.

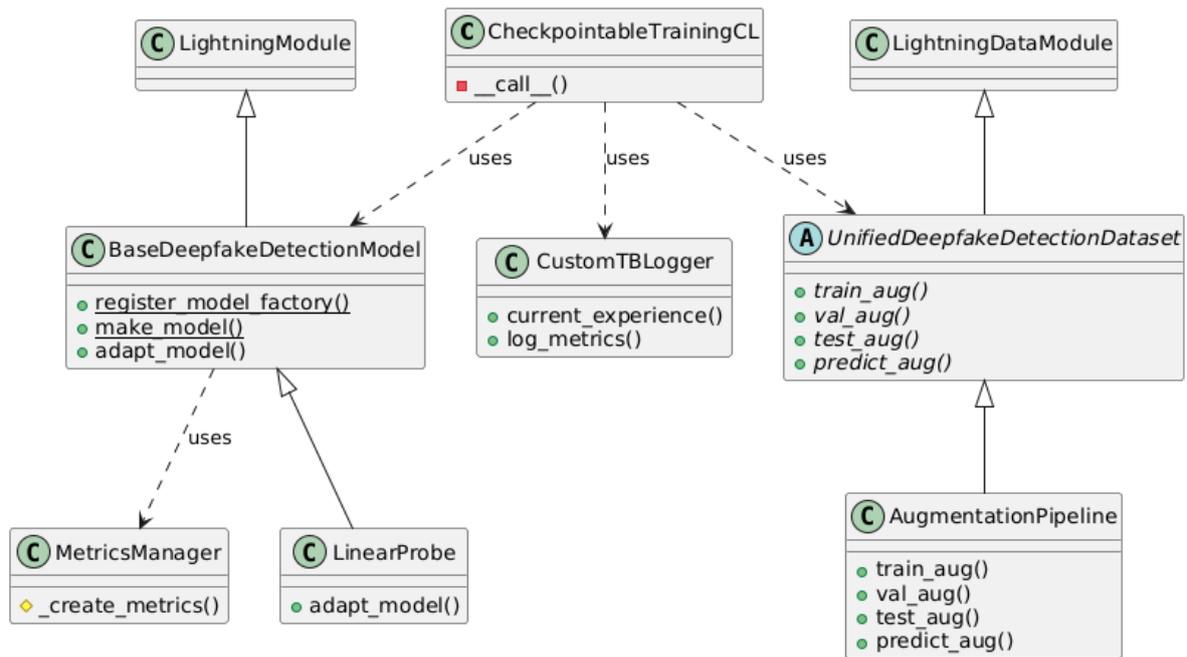


Figura 3.8: Struttura delle classi per l’addestramento e valutazione.

- **Selezione del modello e strategia di addestramento:** utilizzo di tecniche come *fine-tuning* o *linear probing* sui modelli integrati.
- **Metriche di valutazione:** selezione delle metriche da calcolare durante il training, sia quelle standard fornite da PyTorch Lightning, sia metriche personalizzate.
- **Pipeline di pre-processing:** configurazione delle operazioni di trasformazione e data augmentation sui dati in input.

L’elemento centrale dell’addestramento è la classe `CheckpointableTrainingCL` (Figura 3.8), che coordina tutte le operazioni necessarie, interfacciandosi con le seguenti componenti:

- `UnifiedDeepfakeDetectionDataset`: classe astratta che gestisce il database e definisce come applicare la data augmentation e il post-processing. La classe `AugmentationPipeline` che la estende fornisce l’implementazione per le pipeline utilizzate su train, validation e test set.
- `BaseDeepfakeDetectionModel`: gestisce i modelli di deepfake detection e fornisce un’API standard per la registrazione e l’inizializzazione dei modelli. Il comportamento del modello può essere personalizzato tramite le sue sottoclassi, come `LinearProbe`, che congela tutti i pesi e aggiunge un ultimo layer lineare per adattarlo alla classificazione binaria.
- `MetricsManager`: raccoglie e calcola le metriche di valutazione, sia quelle native di PyTorch Lightning, sia metriche personalizzate presenti nel benchmark.
- `CustomTBLogger`: estende le funzionalità di **TensorBoard**[1], aggiungendo il supporto per il continual learning e registrando i valori delle metriche calcolate durante l’addestramento.

Il benchmark, nella sua interezza, garantisce la possibilità di integrare facilmente nuovi modelli, dataset e strategie di training. La modularità delle classi e l'integrazione con PyTorch Lightning facilitano l'estendibilità del benchmark, permettendo l'aggiunta di nuove funzionalità senza modificare il framework di base.





```

)
complete_dataset = DatasetDict(
    {
        "train": concatenate_datasets(
            [unified_fake_dataset["train"], real_dataset["train"]]
        ),
        "validation": concatenate_datasets(
            [unified_fake_dataset["validation"], real_dataset["validation"]]
        ),
    }
)
complete_dataset.save_to_disk(
    OUTPUT_PATH / "complete_dataset_5k_raw",
    num_proc=num_proc if make_jpeg_dataset else None,
    max_shard_size="500MB",
)

```

Listato 4.1: Funzione utilizzata per la costruzione del dataset complessivo.

### 4.1.1 Download dataset pubblici

Per il download dei singoli dataset pubblici utilizzati, sono state adottate diverse tecniche a seconda delle modalità di accesso disponibili. Nella maggior parte dei casi, è stato sufficiente scaricare la repository del dataset o l'archivio corrispondente. Tuttavia, in alcuni casi, la distribuzione del dataset non risultava altrettanto semplice.

Per queste situazioni, in cui il dataset non era fornito in un formato direttamente utilizzabile, si è ricorso prevalentemente alla libreria **img2dataset**, capace di convertire diversi formati di file in un dataset compatibile con la libreria **Datasets**.

Nel caso analizzato (Listato 4.2), il dataset **Polariffshield**[6] viene fornito come un file CSV, in cui ogni riga contiene un prompt, una serie di link alle immagini generate sulla base del prompt, un link per ciascun generatore utilizzato e una serie di informazioni aggiuntive per distinguere le versioni dei generatori impiegati. Dal CSV originale, le informazioni rilevanti sono state estratte mediante la funzione `modify_csv`, producendo un nuovo file CSV compatibile con **Datasets**. Successivamente, tramite la funzione `download`, il nuovo CSV viene utilizzato per scaricare le immagini del dataset, preservando le informazioni relative ai prompt impiegati.

Il risultato finale è un dataset strutturato in cartelle, ognuna dedicata a un generatore specifico. Ogni cartella contiene le immagini corrispondenti, accompagnate da file in formato JSON che conservano i prompt associati alle immagini generate.

```

from typing import Dict, List, Set
from img2dataset import download
import os
import pandas as pd

def modify_csv(input_file: str, output_file: str, columns: Set[str], prefix:
    str):
    df = pd.read_csv(input_file)
    for col in columns:
        if col in df.columns:

```

```

        df[col] = (df[col].astype(str).apply(lambda x: prefix + x if
            pd.notnull(x) else x))
    else:
        print(f"Column '{col}' not found in {input_file}")
df.to_csv(output_file, index=False)

def download_polardiffshield():

    columns_names: Dict[str, List[str]] = {
        "URL firefly": ["ff modelVersion"], "URL midjourney": ["md ver"],
        "URL Dall.e 2": [], "URL Dall.e 3": [], "URL Glide": [],
        "URL Stable Diffusion 1.*": ["sd 1.x model"],
        "URL Stable Diffusion 2.1": [],
        "URL Stable Diffusion XL": [],
    }
    modify_csv("database.csv", "new_database.csv", columns_names.keys(),
              "http://")
    for col, data in columns_names.items():
        output_dir = os.path.abspath(
            col.replace("URL", "")[1:]
            .replace(" ", "-")
            .replace(".", "-")
            .replace("-1-*", "")
        )

        data.append("prompt")
        download(
            url_list="new_database.csv", image_size=-1,
            output_folder=output_dir, processes_count=16,
            thread_count=32, resize_mode="no",
            encode_quality=9, encode_format="png",
            skip_reencode=True, output_format="files",
            input_format="csv", url_col=col,
            save_additional_columns=data,
            number_sample_per_shard=1000,
        )

```

Listato 4.2: Gestione download del dataset Polardiffshield[6].

## 4.1.2 Costruzione dataset sintetici parziali

### Definizione classi principali

Ogni dataset scaricato contenente immagini sintetiche è associato a una specifica classe che estende `DeepfakeDatasetBuilder`. Quest'ultima fornisce le funzionalità comuni a tutti i dataset e definisce alcuni metodi astratti che devono essere implementati nelle classi derivate.

La classe analizzata, `Artifact_Builder` (Listato 4.3), è responsabile della creazione del dataset sintetico per **Artifact**[78], un dataset che raccoglie un vasto numero di immagini generate da molteplici modelli generativi.

```

class Artifact_Builder(DeepfakeDatasetBuilder):
    def __init__(
        self, input_path: PathAlike,
        output_path: PathAlike,
        generator_config: Dict[str, Any],
        convert_to_jpeg: bool = False,
        tmp_cache_dir: Optional[PathAlike] = None,
        num_proc: int = 16, seed: int = 1234,
        check_already_done_marker: bool = True,
        cleanup_cache_on_exit: bool = True,
    ):
        self.input_path = Path(input_path)
        super().__init__(
            output_path=output_path,
            tmp_cache_dir=tmp_cache_dir,
            generator_config=generator_config,
            convert_to_jpeg=convert_to_jpeg,
            seed=seed,
            num_proc=num_proc,
            check_already_done_marker=check_already_done_marker,
            cleanup_cache_on_exit=cleanup_cache_on_exit,
        )

    def _make_dataset(self, indices: List[int]):
        paths = self._make_dataset_paths()
        output_path = paths["output_path"]

        filelist = self._get_image_list()
        filelist = [filelist[i] for i in indices]
        max_samples, lock = create_shared_max_samples(self.generator_config)
        print("Generating the fake images dataset...")
        flattened_subset = Dataset.from_generator(
            generator=partial(
                artifact_test_generator,
                self.input_path,
                FAKE_IMAGES_LABEL,
                self.convert_to_jpeg,
                self.generator_config,
                max_samples,
                lock,
            ),
            features=DEEPFAKE_DATASET_FEATURES,
            num_proc=self.num_proc,
            gen_kwargs={"filelist": filelist},
            cache_dir=(
                str(self.tmp_cache_dir) if self.tmp_cache_dir is not None else
                None
            ),
        )
        print("Saving the fake images dataset to disk:", str(output_path))

```

```

        flattened_subset.save_to_disk(
            str(output_path), num_proc=self.num_proc, max_shard_size="500MB"
        )

    def _dataset_max_len(self) -> int:
        return len(self._get_image_list())

    def _get_image_list(self) -> List[Path]:
        return _artifact_image_list(self.input_path)

    def _make_indices_groups(self) -> Optional[Dict[str, List[int]]]:
        image_list = self._get_image_list()
        indices_dict = defaultdict(list)
        for idx, fpath in enumerate(image_list):
            indices_dict[
                artifact_test_get_generator_name(self.input_path, fpath)
            ].append(idx)
        return indices_dict

```

Listato 4.3: Rappresentazione parziale della classe per la creazione del dataset sintetico parziale a partire da `Artifact`[78].

I metodi astratti ereditati da `DeepfakeDatasetBuilder` e implementati in `Artifact_Builder` sono:

- `_dataset_max_len`: determina la lunghezza del dataset, ovvero il numero totale di immagini disponibili.
- `_get_image_list`: restituisce la lista completa dei percorsi delle immagini pertinenti all'interno del dataset.
- `_make_indices_groups`: organizza le immagini in gruppi in base al modello di generazione impiegato.

### Raccolta e raggruppamento dei percorsi delle immagini

Ogni dataset utilizzato presenta una struttura unica nell'organizzazione delle immagini, rendendo necessaria l'implementazione di funzioni specifiche per ciascuno. Tuttavia, queste funzioni svolgono gli stessi compiti o compiti simili, come la raccolta e l'ordinamento delle immagini per garantire coerenza e riproducibilità.

In particolare, la funzione `_artifact_image_list` (Listato 4.4) si occupa di raccogliere tutti i percorsi delle immagini sintetiche presenti nel dataset `Artifact`, ordinandole alfabeticamente per garantire il determinismo nella selezione delle immagini.

Per ottimizzare le prestazioni e ridurre il tempo di accesso ai dati, la funzione utilizza il decoratore `@lru_cache(maxsize=1)`, che memorizza il risultato della funzione in una cache per evitare letture ripetitive dei file su disco. Questo meccanismo consente di migliorare l'efficienza nelle operazioni di accesso ai percorsi delle immagini.

```

@lru_cache(maxsize=1)
def _artifact_image_list(folder: Path) -> List[Path]:
    all_paths = []
    for folder in tqdm(list(folder.iterdir())):

```

```

if not folder.is_dir():
    continue
current_paths = set()
csv_metadata_path = folder / "metadata.csv"
with open(csv_metadata_path, "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        img_path = folder / row["image_path"]
        if int(row["target"]) != 0:
            current_paths.add(img_path)
all_paths.extend(current_paths)
return sort_paths(all_paths)

```

Listato 4.4: Funzione di supporto alla classe `Artifact_Builder` per la raccolta dei percorsi delle immagini.

Il numero di immagini, l'elenco ordinato dei percorsi e i gruppi di immagini per generatore sono utilizzati dai metodi generali definiti in `DeepfakeDatasetBuilder`, validi per tutti i dataset.

### Selezione immagini

Tra questi, il più rilevante è `_select_subset_indices` (Listato 4.5), che sfrutta i metodi implementati in `Artifact_Builder` per selezionare gli indici delle immagini da includere nella costruzione del dataset sintetico.

La selezione degli indici avviene in base alla dimensione del sottoinsieme e alla presenza di gruppi di immagini per generatore, ovvero immagini dello stesso generatore ma divise per soggetto rappresentato. Se è attiva l'opzione `use_samples_per_generator`, il metodo distribuisce le immagini in base alla quantità di campioni specificata per ciascun modello generativo.

Gli indici così ottenuti vengono poi utilizzati dal metodo `_make_dataset` di `Artifact_Builder` (Listato 4.3), che gestisce la costruzione del dataset attraverso una funzione parziale e ne salva il contenuto su disco.

```

def _select_subset_indices(self) -> List[int]:
    seed = self._make_seed()
    self._init_seed(seed)
    max_len = self._dataset_max_len()
    if max_len == 0:
        return []
    elif self.subset_size < 0 or self.subset_size >= max_len:
        return list(range(max_len))
    else:
        indices_groups_dict = (
            self._make_indices_groups()
            if self.indices_groups is None
            else self.indices_groups
        )
        if indices_groups_dict is not None:
            selected_indices = []
            group_names = list(indices_groups_dict.keys())

```

```
indices_groups = list(indices_groups_dict.values())
if self.use_samples_per_generator:
    new_indices_groups_dict = self.__map_groups_to_generator(
        indices_groups_dict
    )
    for generator, samples in self.samples_per_generator.items():
        selected_indices.extend(
            random.sample(
                new_indices_groups_dict[generator],
                min(samples,
                    len(new_indices_groups_dict[generator])),
            )
        )
    return selected_indices
```

Listato 4.5: Rappresentazione parziale del metodo `_select_subset_indices` per la selezione delle immagini.

### Creazione del dataset

Tramite gli indici selezionati viene creato il dataset di immagini sintetiche sfruttando la funzione `from_generator` di **Datasets** e la funzione `artifact_test_generator` (Listato 4.6). Questa funzione itera sui file in batch grazie all'uso della classe `MultiSourceFilesIterator`, un iteratore ottimizzato per gestire file provenienti da più sorgenti, incluse directory locali e archivi compressi. Questo meccanismo riduce il numero di accessi a disco e migliora le prestazioni estraendo gruppi di file in una directory temporanea prima di restituirli uno alla volta.

Durante l'iterazione, `artifact_test_generator` esegue diversi controlli sulle immagini:

1. Prova ad aprire ogni file con `Image.open(fpath)`, scartando quelli che generano eccezioni.
2. Filtra le immagini in base alla dimensione, eliminando quelle inferiori a  $128 \times 128$  o superiori a  $8192 \times 8192$  pixel. Questo filtro è fissato per escludere da un lato le immagini troppo piccole per essere valide e dall'altro le immagini troppo grandi che richiedono molto tempo di elaborazione.
3. Recupera i metadati rilevanti presenti nel dataset originale (Sezione 3.4.1).

La funzione fa uso del `for-yield` così un generatore, evitando di caricare in memoria l'intero dataset contemporaneamente. Il generatore viene consumato direttamente da `from_generator`, che costruisce il dataset in modo efficiente, iterando dinamicamente sui campioni senza doverli mantenere tutti in RAM. Questo approccio è particolarmente utile quando si lavora con grandi volumi di dati, migliorando sia il tempo di esecuzione che l'uso della memoria.

```

def artifact_test_generator(root_path: Path, label: int,
    convert_to_jpeg: bool, generator_config: Dict[str, Any],
    max_samples, lock, filelist: List[Path],
):
    with MultiSourceFilesIterator(filelist) as file_iter:
        for fpath in file_iter:
            generator = artifact_test_get_generator_name(root_path, fpath)
            if not check_needed_samples(max_samples, generator, lock):
                continue
            try:
                img = Image.open(fpath)
            except Exception as e:
                print(f"Problematic image: {fpath} produce error {e}")
                continue
            if not is_image_valid(img):
                print(f"broken img: {fpath}")
                continue
            relative_path = fpath.relative_to(root_path)
            conditioning = None
            description = None
            positive_prompt = None
            paired_real_images = []
            if not check_max_samples(max_samples, generator, lock):
                continue
            img = prepare_image(img, convert_to_jpeg=convert_to_jpeg)
            context = DatasetContext()
            context.add_values(
                {
                    "parent_name": lambda: fpath.parent.name,
                    "parent_parent_name": lambda: fpath.parent.parent.name,
                    "cips_description": lambda ctx=context: ctx[
                        "parent_name"
                    ].removeprefix("cips-"),
                    "class_id": lambda ctx=context: ctx["parent_name"],
                    "cyclegan_description": lambda ctx=context: ctx[
                        "parent_name"
                    ].split("2")[-1],
                    "diffusiongan_description": lambda ctx=context: ctx[
                        "parent_parent_name"
                    ]
                }
                .removeprefix("lsun-")
                .removesuffix("-data"),
                "ganformer_description": lambda ctx=context: ctx[
                    "parent_name"
                ].removesuffix("_images"),
                "stargan_face_modifier": lambda ctx=context:
                    ctx["parent_name"]
                .replace("_", " ")
                .lower(),
            )

```

```

        "stablediffusion_face_modifier": lambda ctx=context: ctx[
            "parent_name"
        ].lower(),
        "stylegan2_description": lambda:
            relative_path.parts[1].split("-")[
                0
            ],
        "stylegan3_description": lambda ctx=context: ctx[
            "parent_name"
        ].split("-")[-2],
        "vqgan_base": lambda: relative_path.parts[1],
        "imagenet_class": lambda: IMAGENET_CLASS_ID_TO_STR[
            int(fpath.parent.name)
        ],
    },
)
data = {
    "fpath": fpath,
}
values = ["conditioning", "description", "positive_prompt"]
final_values = process_generator(
    generator_config, generator, values, context, data
)
conditioning = final_values["conditioning"]
description = final_values["description"]
positive_prompt = final_values["positive_prompt"]
yield {
    "image": img,
    "label": label,
    "generator": generator,
    "filename": str(relative_path),
    "description": description,
    "positive_prompt": positive_prompt,
    "negative_prompt": "",
    "conditioning": conditioning,
    "origin_dataset": "Artifact",
    "paired_real_images": paired_real_images,
    "found_in_datasets": ["Artifact"],
}

```

Listato 4.6: Definizione parziale della funzione `artifact_test_generator`.

### Configurazione parametri e raccolta dei metadati

Data la necessità di specificare con precisione il numero di immagini da utilizzare per ogni gruppo di immagini generate dallo stesso modello, ogni dataset è stato associato a un documento YAML di configurazione (Listato 4.7). Questo documento definisce tutte le informazioni necessarie per la gestione del numero di immagini per generatore e la raccolta dei metadati da includere nel database finale.

Per la selezione del numero di immagini richiesto, sono disponibili tre modalità:

1. **take all:** Se la chiave `sampling.take_all`, se è impostata su `True`, vengono selezionate tutte le immagini di tutti i generatori. Questa opzione è utilizzata unicamente per dataset di piccole dimensioni con una distribuzione non uniforme delle immagini tra i generatori o per finalità di testing sul contenuto del dataset.
2. **distribute:** utilizzata per dataset di piccole dimensioni in cui ogni generatore ha lo stesso numero di immagini selezionate.
3. **custom:** la modalità più comune, che permette di specificare manualmente il numero di immagini per ogni generatore attraverso la chiave `samples`.

Nel caso in cui siano definite più modalità contemporaneamente, la priorità viene assegnata nell'ordine sopra indicato (`take all > distribute > custom`).

Oltre al numero totale di immagini per generatore, è possibile definire, tramite la chiave `max_samples`, il numero massimo di immagini da considerare per ciascun generatore. Questo valore, generalmente inferiore a quello indicato in `samples`, serve a creare un margine di sicurezza per escludere eventuali immagini corrotte o non conformi alle specifiche dimensionali.

Oltre a specificare il numero di immagini desiderato per ogni generatore tramite la chiave `samples`, è possibile impostare un limite massimo di immagini da considerare con la chiave `max_samples`. Generalmente, `max_samples` è inferiore a `samples` e ha lo scopo di garantire la qualità del dataset. Se l'insieme di immagini selezionato con `samples` include elementi corrotti o non adeguati, questi vengono scartati. La chiave `max_samples` assicura che, dopo questo filtraggio, il numero di immagini rimanenti sia esattamente pari al suo valore, eliminando eventuali immagini in eccesso e generando un warning in caso di immagini insufficienti.

```
sampling:
  take_all: False

CIPS:
  samples: 5500
  max samples: 5000
  conditioning:
    value: "noise"
  description:
    value: "{cips_description}"
  mappings:
    - match: "landscape"
      value: "landscape"
    - match: ["churces", "churches"]
      value: "church"
    - match: "ffhq"
      value: "face"
  positive_prompt:
    value: ""
CycleGAN:
  samples: 4400
  max samples: 4000
  conditioning:
    value: "image/{parent_name}"
```

```
description:
  value: "{cyclegan_description}"
  mappings:
    - match: "monet"
      value: "Monet painting"
  positive_prompt:
    value: ""
Glide:
  samples: 1800
  max samples: 1600
  conditioning:
    value: !artifact glide_conditioning
  description:
    value: ""
  positive_prompt:
    value: ""
```

Listato 4.7: Aspetti principali della configurazione per il dataset Artifact.

Per ciascun generatore, nel documento YAML di configurazione, sono definite le modalità per recuperare i metadati opzionali associati a ciascuna immagine. Tali metadati possono essere ottenuti in modalità diverse a seconda che si tratti di valori costanti o di valori dinamici che variano in funzione dell'immagine.

Nel caso di un metadato costante, è sufficiente assegnarlo direttamente come valore della chiave. Ad esempio, nel dataset Artifact, per il generatore CIPS[4] il campo `conditioning` assume sempre il valore "noise" per tutte le immagini. Al contrario, se il metadato è variabile, è possibile specificare il nome di una funzione lambda definita all'interno di una mappa (ad esempio, il dizionario `context` illustrato nel Listato 4.6 raccoglie tutte le funzioni per il dataset Artifact). In questo caso, il valore verrà calcolato solo quando richiesto.

Spesso, i metadati presenti nel dataset originale non sono immediatamente utilizzabili, poiché non sono stati progettati per un uso diretto. Per facilitare la standardizzazione di questi dati, la configurazione YAML consente di mappare i valori recuperati a nuovi valori predefiniti tramite una lista di `mappings`. Questo meccanismo permette di specificare una serie di trasformazioni: se il valore calcolato coincide con uno dei criteri definiti nei `mappings`, esso viene automaticamente sostituito dal valore corrispondente.

Ad esempio, sempre per il generatore CIPS, il campo `description` viene impostato richiamando la funzione lambda identificata da `cips_description`; se il valore ottenuto corrisponde a uno dei criteri del mapping associato, esso viene sostituito dal valore predefinito indicato.

Infine, qualora alcuni metadati non possano essere recuperati con le tecniche sopra descritte, è possibile fare riferimento a una funzione personalizzata, definita nel codice, che calcola il metadato necessario. Ad esempio, per il generatore Glide[64] il valore del campo `conditioning` viene determinato mediante la funzione `glide_conditioning` presente nel codice.

Per la gestione complessiva dei metadati associati alle immagini, sono utilizzate due funzioni principali: `process_generator` e `resolve_mapping` (Listato 4.8). La funzione `process_generator` è responsabile di processare la configurazione di un generatore specifico, come definita nel file YAML. Essa analizza ciascun campo (ad esempio, `conditioning`, `description`) e valuta i relativi valori. In particolare, se il valore è defi-

nito come una stringa, viene formattato con il dizionario di contesto tramite il metodo `format`; se invece è una funzione (callable), essa viene eseguita con i dati aggiuntivi forniti. Il risultato di questa prima fase è un insieme di valori grezzi, che possono contenere placeholder da risolvere ulteriormente.

A completamento, la funzione `resolve_mapping` si occupa del processo di mapping: essa confronta il valore risolto con una serie di regole definite nella chiave `mappings` e, se il valore soddisfa una determinata condizione (ad esempio, corrispondenza esatta o tramite espressione regolare), lo sostituisce con un nuovo valore definito nel mapping. Questo meccanismo supporta anche mapping ricorsivi, permettendo una standardizzazione approfondita dei metadati. Il risultato finale è un dizionario che associa, per ciascun campo del generatore, il valore definitivo da utilizzare nel database.

```
def resolve_mapping(
    resolved_value: str, mappings: List[Dict[str, Any]], context: Dict[str, Any]
) -> str:
    for mapping in mappings:
        match = mapping.get("match")
        if (isinstance(match, list) and resolved_value in match) or (
            isinstance(match, str)
            and (resolved_value == match or re.fullmatch(match, resolved_value))
        ):
            resolved_value = mapping["value"].format(**context)
            nested_mappings = mapping.get("mappings", [])
            if nested_mappings:
                resolved_value = resolve_mapping(
                    resolved_value, nested_mappings, context
                )
            break
    return resolved_value

def process_generator(
    generator_configs: Dict[str, Any],
    generator: str,
    values: List[str],
    context: Dict[str, Any] = {},
    data: Dict[str, Any] = {},
) -> Dict[str, Any]:
    config = generator_configs.get("generators", {}).get(generator)
    if not config:
        raise ValueError(f"Unknown generator: {generator}")
    final_values = {}
    for value in values:
        field_config = config.get(value, {})
        config_value = field_config.get("value")
        if callable(config_value):
            resolved_value = config_value(data)
        elif isinstance(config_value, str):
            resolved_value = config_value.format(**context)
        else:
            resolved_value = config_value
    mappings = field_config.get("mappings", [])
```

```

    resolved_value = resolve_mapping(resolved_value, mappings, context)
    final_values[value] = resolved_value
return final_values

```

Listato 4.8: Funzioni a supporto della valutazione e del mapping dei metadati.

### 4.1.3 Costruzione dataset reale

La gestione del dataset di immagini reali avviene in modo differente rispetto a quella del dataset sintetico, principalmente perché non è necessario separare le immagini per generatore e molti dei metadati tipici delle immagini sintetiche non sono presenti.

A tale scopo, è stata definita la classe astratta `RealImagesBuilder`, che modella un dataset generico di immagini reali senza implementare direttamente i metodi necessari. Analogamente a quanto avviene con `DeepfakeDatasetBuilder` per i dataset sintetici, le sottoclassi di `RealImagesBuilder` forniscono le implementazioni specifiche per ciascun dataset reale. In particolare, i metodi astratti definiti sono:

- `get_builder_name`: restituisce il nome del dataset reale.
- `get_prefix`: indica quale parte del dataset (ad esempio, train, validation o test) viene considerata, utile quando il dataset originale è partizionato e se vuole utilizzare soltanto una porzione.
- `available_images`: individua e raccoglie tutti i percorsi delle immagini disponibili.
- `get_image`: processa il percorso di un'immagine e restituisce un oggetto di tipo `RowDict` che rappresenta una singola entry del dataset, con i relativi metadati.

Infine, il metodo `select_random_images` definisce, quando implementato, una strategia di selezione delle immagini, alternativa a quella casuale.

```

class RealImagesBuilder(ABC):

    @abstractmethod
    def get_image(
        self,
        image_id: str,
        found_in_datasets: Optional[List[str]] = None,
        convert_to_jpeg: bool = False,
    ) -> RowDict:
        pass

    @abstractmethod
    def available_images(self) -> Iterable[str]:
        pass

    @abstractmethod
    def get_prefix(self) -> str:
        pass

```

```

@abstractmethod
def get_builder_name(self) -> str:
    pass

def select_random_images(
    self,
    num_images: int,
    seed: Optional[int] = None,
    excluding: Optional[Iterable[str]] = None,
    allowed: Optional[Iterable[str]] = None,
) -> Optional[List[str]]:
    return None

```

Listato 4.9: Definizione della classe astratta RealImagesBuilder.

Tutte le istanze delle classi che implementano RealImagesBuilder sono gestite dalla classe RealImagesManager (Listato 4.10), che coordina l'utilizzo dei vari dataset reali. Le sue funzioni principali, quali get\_image e available\_images, sfruttano il prefisso identificativo del dataset (estratto dall'identificatore dell'immagine) per delegare il recupero delle informazioni all'istanza corrispondente di RealImagesBuilder tramite metodi omonimi.

Il metodo select\_random\_images definisce la strategia generale per la selezione casuale delle immagini. In particolare, per ciascun builder vengono raccolte le immagini disponibili secondo le specifiche strategie di selezione implementate a livello individuale, e successivamente viene applicata la funzione saturating\_balanced\_choice. Questa funzione garantisce che la distribuzione percentuale delle immagini selezionate rispecchi quella delle immagini disponibili in ciascun dataset, mantenendo così la proporzione originaria.

Nel processo di costruzione del dataset reale, saturating\_balanced\_choice viene utilizzata in due fasi: la prima per selezionare le immagini associate alle controparti sintetiche, e la seconda per raggiungere un numero complessivo di immagini pari a quello del dataset sintetico, assicurando così un bilanciamento equilibrato tra le classi reali e sintetiche.

```

class RealImagesManager:
    def __init__(self, real_images_builders: List[RealImagesBuilder]):
        self.real_images_builders = real_images_builders
        self._builder_map: Dict[str, RealImagesBuilder] = dict()
        self._make_builder_map()

    def available_images(self, builder_prefix: Optional[str] = None,
        among: Optional[Iterable[str]] = None) -> Iterable[str]:
        among = set(among) if among is not None else None
        if builder_prefix is not None:
            return [
                image_id
                for image_id in
                    self._builder_map[builder_prefix].available_images()
                if among is None or image_id in among
            ]
        else:

```

```

    if among is not None:
        all_available_images = set()
        for builder in self.real_images_builders:
            all_available_images.update(builder.available_images())
        return all_available_images.intersection(among)
    else:
        return [
            image_id
            for builder in self.real_images_builders
            for image_id in builder.available_images()
        ]

def get_image(self, image_id: str,
              found_in_datasets: Optional[List[str]] = None,
              convert_to_jpeg: bool = False,
) -> RowDict:
    builder_prefix = image_id.split("/")[0]
    return self._builder_map[builder_prefix].get_image(
        image_id,
        found_in_datasets=found_in_datasets,
        convert_to_jpeg=convert_to_jpeg,
    )

def available_builder_prefixes(self):
    return list(self._builder_map.keys())

def _make_builder_map(self):
    for builder in self.real_images_builders:
        builder_prefix = builder.get_prefix()
        self._builder_map[builder_prefix] = builder

def select_random_images(self, num_images: int, seed: int,
                        excluding: Optional[Iterable[str]] = None,
                        allowed_prefixes: Optional[Iterable[str]] = None,
) -> List[str]:
    allowed_prefixes = (
        set(allowed_prefixes)
        if allowed_prefixes is not None
        else set(self._builder_map.keys())
    )
    excluding = set(excluding) if excluding is not None else set()
    allowed_builders = [
        builder
        for prefix, builder in self._builder_map.items()
        if prefix in allowed_prefixes
    ]
    allowed_images: Dict[RealImagesBuilder, List[str]] = dict()
    for builder in allowed_builders:
        allowed_images[builder] = [
            x for x in builder.available_images() if x not in excluding
        ]

```

```

        allowed_images[builder].sort()
possible_choices: List[str] = []
associated_builders: List[RealImagesBuilder] = []
for builder, images in allowed_images.items():
    possible_choices.extend(images)
    associated_builders.extend([builder] * len(images))
return saturating_balanced_choice(
    num_images,
    possible_choices,
    associated_builders,
    seed,
    choice_fn=_random_choice_fn,
)[0]

```

Listato 4.10: Definizione della classe RealImagesManager.

## 4.2 Configurazione, pre-processing e Modelli

### 4.2.1 Configurazione degli iperparametri

Per l'addestramento dei modelli si utilizza come configurazione un file YAML che definisce tutti gli iperparametri e le impostazioni necessarie, suddivise in 3 sezioni principali: **trainer**, **data** e **model**.

#### Trainer

La sezione trainer (Listato 4.11) definisce l'intero processo di addestramento del modello, stabilendo le modalità con cui il benchmark gestisce le operazioni computazionali e la distribuzione dei task. In particolare, alcuni iperparametri permettono di definire l'hardware su cui eseguire il modello (*devices*) e la strategia di distribuzione per parallelizzare i calcoli su più dispositivi (*strategy*). Inoltre, è possibile specificare la precisione numerica dei calcoli, garantendo che le operazioni avvengano con la precisione adatta per l'addestramento (*precision*).

All'interno della configurazione del trainer è presente una sezione dedicata al **logger**, che si occupa di registrare in modo dettagliato le informazioni e gli eventi che si verificano durante il training. Il sistema di logging si basa sull'utilizzo di una classe personalizzata, nell'esempio `CustomTBLogger`, e permette di salvare i log in una cartella specifica, organizzare i nomi degli esperimenti con prefissi dedicati e configurare aspetti come il salvataggio del grafo computazionale e il tracciamento delle metriche degli iperparametri, in modo da poter gestire in modo organizzato gli addestramenti eseguiti.

Oltre a questi aspetti, la configurazione del trainer include numerosi parametri che controllano la dinamica dell'addestramento: ad esempio, esistono impostazioni per eseguire run di debug rapide (*fast\_dev\_run*), per definire il numero di epoche o step che l'addestramento deve coprire, e per gestire eventuali limitazioni sul numero di batch utilizzati nelle diverse fasi (*train*, *validation*, *test*, *predict*). Tra questi una impostazione fondamentale è il salvataggio periodico dei checkpoint del modello, che consente di riprendere l'addestramento in caso di interruzioni o per analisi successive.

La sezione **callbacks**, invece, definisce una serie di funzioni che vengono eseguite in momenti specifici del ciclo di addestramento. Queste callback sono progettate per monitorare e registrare aspetti cruciali del training, come ad esempio il learning rate, e per gestire il logging delle metriche relative alle prestazioni del modello. Alcune callback sono dedicate a particolari task, come il salvataggio delle predizioni relative a specifici obiettivi, mentre altre sono pensate per gli addestramenti a Sliding Window e Continual Learning, utilizzando differenti meccanismi di calcolo delle metriche per offrire una valutazione del modello durante tutto il processo di addestramento.

```
trainer:
  accelerator: auto
  strategy: auto
  devices: auto
  precision: "32-true"
  logger:
    class_path: training_utils.CustomTBLogger
    init_args:
      save_dir: "./ddset_results"
      name_prefix: "experiment_"
      experience_id: null
      log_graph: false
      default_hp_metric: false
      prefix: ''
      sub_dir: null
      experiment_id: null
      comment: ''
      purge_step: null
      max_queue: 10
      flush_secs: 120
      filename_suffix: ''
  fast_dev_run: false
  max_epochs: 1
  min_epochs: 1
  max_steps: -1
  min_steps: null
  max_time: null
  limit_train_batches: null
  limit_val_batches: null
  limit_test_batches: null
  limit_predict_batches: null
  overfit_batches: 0.0
  val_check_interval: null
  check_val_every_n_epoch: 1
  num_sanity_val_steps: 0
  log_every_n_steps: null
  enable_checkpointing: true
  enable_progress_bar: null
  enable_model_summary: null
  accumulate_grad_batches: 1
  gradient_clip_val: 1
  gradient_clip_algorithm: null
```

```

deterministic: null
benchmark: null
inference_mode: true
use_distributed_sampler: true
profiler: null
detect_anomaly: false
barebones: false
plugins: null
sync_batchnorm: false
reload_data loaders_every_n_epochs: 0
default_root_dir: null
callbacks:
  - class_path: training_metrics.DeepFakePredictionsDump
  - class_path: lightning.pytorch.callbacks.LearningRateMonitor
  - class_path: training_metrics.MetricsManager
  - class_path: training_metrics.MetricsManagerCL
    init_args:
      compute_mechanism: "immediate_future"
  - class_path: training_metrics.MetricsManagerCL
    init_args:
      compute_mechanism: "growing"
  - class_path: training_metrics.MetricsManagerCL
    init_args:
      compute_mechanism: "growing_whole"

```

Listato 4.11: Esempio di configurazione iperparametri della sezione **trainer**.

## Data

La sezione data (Listato 4.12) definisce il modulo responsabile del caricamento e della trasformazione dei dati tramite una classe dedicata, nel caso in esempio viene utilizzata `ClipBasedSID`, che gestisce l'intero flusso dei dati necessari per l'addestramento, la validazione e il test del modello.

Gli iperparametri definiti permettono di configurare aspetti fondamentali della preparazione dei dati. Ad esempio, sono definiti i batch size per training e validation (`train_batch_size` e `eval_batch_size`), la dimensione a cui saranno portate le immagini prima di iniziare l'addestramento (`image_size`) e il numero di worker, consentendo di sfruttare la parallelizzazione per accelerare il caricamento e il pre-processing dei dati (`num_workers`). Ulteriori iperparametri permettono di controllare la quantità di dati utilizzata, per poter predisporre addestramenti su dataset ridotti (`train_subset_size` e `validation_subset_size`) e la dimensione massima delle immagini sopra la quale le immagini vengono forzatamente ridimensionate prima della data augmentation per evitare problemi di prestazioni o consumo eccessivo di risorse (`maximun_image_size`).

Per l'augmentation la configurazione prevede la possibilità di garantire trasformazioni deterministiche (`deterministic_augmentations`), in modo che ogni esecuzione produca risultati riproducibili. Inoltre, viene definito un fattore di augmentation (`augmentation_factor`), che specifica quante varianti di ogni immagine originale devono essere generate.

La configurazione include anche una sezione dedicata al benchmark nel contesto delle Sliding Window e del Continual Learning. Qui vengono definiti il numero di generatori per ogni esperienza (`n_generators_per_experience`), cioè su quanti generatori viene addestrato il modello in ogni fase dell'apprendimento, e la strategia utilizzata per il tipo di benchmark utilizzato (`benchmark_type`): "cumulative" per le Sliding Window e "cl" per il Continual Learning.

```
data:
  class_path: lightning_data_modules.ClipBasedSID
  init_args:
    deepfake_dataset_dir: "/deepfake/deepfake_benchmark_raw"
    train_batch_size: 256
    eval_batch_size: 256
    image_size: 224
    num_workers: 16
    train_subset_size: 0
    validation_subset_size: 0
    deterministic_augmentations: true
    augmentation_factor: 4
    augmentations_base_seed: 4321
    maximum_image_size: 1080
    cl_becnhmark_definition:
      n_generators_per_experience: 4
      current_experience: null
      benchmark_type: "cumulative"
```

Listato 4.12: Esempio di configurazione iperparametri della sezione **data**.

## Model

La sezione `model` (Listato 4.13) definisce l'architettura e le impostazioni relative al modello da addestrare. Anche in questa sezione, come nelle precedenti, si possono utilizzare classi apposite che definiscono come gestire il modello, nell'esempio si utilizza la classe `LinearProbe` che consiste nell'applicare un classificatore lineare su una backbone pre-addestrata, in questo caso il modello stesso. Il modello (`model_name`) specificato viene utilizzato per identificare la classe relativa all'implementazione del modello.

All'interno della configurazione del modello, la parte dedicata all'ottimizzazione stabilisce quale algoritmo impiegare per aggiornare i pesi durante il training, sempre specificato tramite la relativa classe per permettere l'uso di ottimizzatori personalizzati. Sono poi specificati gli argomenti per inizializzare l'ottimizzatore, tra questi, il `learning rate` (`lr`) controlla la velocità con cui il modello apprende inizialmente dai dati, `momentum`[74] e `nesterov`[62] per facilitare la convergenza e la stabilità del modello.

La configurazione prevede anche l'impiego di uno scheduler, nell'esempio viene utilizzato il **OneCycleLR**[87], una strategia di scheduling in cui il learning rate parte da un valore basso, aumenta fino a raggiungere un picco e poi decresce rapidamente fino a un valore molto piccolo. Infine, l'iperparametro dedicato alla soglia di classificazione (`classification_treshold`) definisce la soglia utilizzata per tradurre le probabilità di output in decisioni binarie, nel caso del benchmark, immagini reali o sintetiche.

```
model:
```

```

class_path: algorithms.LinearProbe
init_args:
  model_name: "RN50"
  optimizer:
    class_path: torch.optim.SGD
    init_args:
      lr: 0.001
      momentum: 0.9
      dampening: 0.0
      weight_decay: 0.0001
      nesterov: false
      maximize: false
      foreach: null
      differentiable: false
  scheduler: "OneCycleLR"
classification_threshold: 0.5

```

Listato 4.13: Esempio di configurazione iperparametri della sezione **model**.

## 4.2.2 Pipeline e tecniche di data augmentation

Per la gestione della data augmentation sono state predefinite due pipeline distinte, una per il training set e una per il validation e il test set. In particolare, la pipeline per il validation e test set (Listato 4.14) applica una serie di trasformazioni con la finalità di rendere le immagini quanto più simili possibili a quelle che sarebbero reperibili in un contesto reale, come immagini reperibili sul web.

Le trasformazioni che contribuiscono maggiormente al raggiungimento del risultato desiderato sono quelle di compressione, realizzate tramite la funzione `data_augment_cmp` e di crop tramite la classe `RandomResizedCropVariable`.

La compressione è infatti caratteristica delle immagini presenti online, che ogni volta che sono scaricate e ripubblicate, a seconda del loro formato, subiscono una compressione. Il modello deve quindi apprendere come riconoscere correttamente anche questa tipologia di immagini, di conseguenza nella pipeline si utilizzano tre compressioni con probabilità decrescente per simulare questo aspetto.

La classe `RandomResizedCropVariable` permette di applicare una trasformazione che, presa la singola immagine, ne ridimensiona una porzione a un valore di altezza e larghezza casuale. L'utilizzo di un `resize/crop` come trasformazione dei dati è altrettanto cruciale per addestrare classificatori robusti, soprattutto in un contesto in cui le immagini online vengono frequentemente ritagliate, ridimensionate o presentano solo una porzione della scena originale. Questa tecnica permette di simulare le variazioni intrinseche derivanti dai processi di pubblicazione e condivisione delle immagini, esponendo il modello a diverse scale e inquadrature.

Le restanti trasformazioni, conversione in scala di grigi e rumore gaussiano, si combinano alla compressione e al `resize/crop` per introdurre ulteriore variabilità.

```

def val_aug(self):
    crop_if_big = transforms.Identity()
    if self.maximum_image_size is not None:
        crop_if_big = RandomCropIfLarge(

```

```
        threshold=(self.maximum_image_size, self.maximum_image_size),
        force_central_crop=True,
    )
    return transforms.Compose(
        [
            crop_if_big,
            transforms.RandomGrayscale(p=0.1),
            transforms.RandomApply(
                [
                    transforms.Lambda(
                        lambda img: data_augment_cmp(img, ["cv2", "pil"],
                                                    [50, 101])
                    )
                ],
                p=0.50,
            ),
            transforms.RandomApply(
                [
                    self._wrap_albumentation(
                        GaussNoise(std_range=(0.015, 0.075), p=1.0)
                    )
                ],
                p=0.3,
            ),
            transforms.RandomApply(
                [
                    transforms.Lambda(
                        lambda img: data_augment_cmp(img, ["cv2", "pil"],
                                                    [50, 101])
                    )
                ],
                p=0.20,
            ),
            transforms.RandomApply(
                [
                    RandomResizedCropVariable(
                        min_size=256,
                        max_size=512,
                        scale=(0.5, 1.0),
                        ratio=(0.9, 1 / 0.9),
                        interpolation_options=[
                            InterpolationMode.BILINEAR,
                            InterpolationMode.BICUBIC,
                            InterpolationMode.LANCZOS,
                        ],
                    )
                ],
                p=0.25,
            ),
            transforms.RandomApply(
                [
```

```

        transforms.Lambda(
            lambda img: data_augment_cmp(img, ["cv2", "pil"],
                [50, 101])
        )
    ],
    p=0.10,
),
transforms.ToImage(),
transforms.ToDtype(torch.float32, scale=True),
transforms.Normalize(
    mean=(0.48145466, 0.4578275, 0.40821073),
    std=(0.26862954, 0.26130258, 0.27577711),
),
]
)

```

Listato 4.14: Pipeline per la data augmentation del validation set.

### 4.2.3 Costruzione del modello

Ogni modello presente nel benchmark è rappresentato da una sua classe che ne modella l'architettura e la gestione. Queste classi, che estendono `Torch.nn.Module` per permetterne l'interoperabilità con il resto del benchmark, sono caratterizzate da due metodi principali: `__init__` dove il modello viene caricato e adattato alla task di classificazione e `forward` che definisce il percorso dei dati nel modello. I modelli sono registrati in un dizionario nella classe `BaseDeepfakeDetectionModel` utilizzando come chiave il nome del modello da utilizzare nella configurazione dell'addestramento.

Uno dei modelli integrati nel benchmark durante la tesi è `DINOv2`[69] tramite la classe `DINOv2Model` (Listato 4.15). Questa classe rappresenta un esempio di transfer learning in cui il modello utilizzato viene congelato e gli viene aggiunto un classificatore lineare per adattarlo alla classificazione. In questo modo è possibile utilizzare le capacità di feature extraction di `DINOv2` evitando il costo computazionale di un addestramento da zero.

```

def make_dinov2_model(model_name: str, pretrained: bool = True, **kwargs):
    try:
        return DINOv2Model(model_name)
    except Exception as e:
        return None

class DINOv2Model(torch.nn.Module):
    def __init__(self, model_name: str, num_classes: int = 1, shape=(3, 224,
        224)):
        super().__init__()
        self.model = torch.hub.load("facebookresearch/dinov2", model_name)
        self.model.eval()
        self.model.requires_grad_(False)
        with torch.no_grad():
            dummy_input = torch.zeros((1, *shape))
            features: Tensor = self.model(dummy_input)

```

```

        self.intermediate_size: int = features.shape[-1]

        self.fc = Linear(self.intermediate_size, num_classes)

    def forward(self, x: Tensor, return_feature=False):
        features = self.model(x)
        if return_feature:
            return features
        return self.fc(features)

BaseDeepfakeDetectionModel.register_model_factory("dinov2", make_dinov2_model)

```

Listato 4.15: Registrazione e architettura di DINOv2[69].

Un altro modello implementato in questo lavoro di tesi è una variante di JanusPro[17], nella classe JanusImageEncoderTune (Listato 4.16). Il modello implementato contiene solo la parte del feature encoder di Janus, ma, a differenza di DINOv2, qui l'intero modello è completamente addestrabile. In particolare, su questa specifica implementazione del modello, viene effettuato fine tuning, ovvero i pesi del modello vengono aggiornati durante l'addestramento per adattarsi meglio all'obiettivo.

Il modello è diviso in due parti: la parte di pre-processing dell'immagine e la parte del feature encoder. La prima, utilizzata nel modello completo per processare in modo combinato il prompt e le immagini associate, viene adattata e utilizzata unicamente per elaborare le immagini in modo che il risultato sia comunque utilizzabile dal feature encoder. La seconda è l'effettiva estrazione delle feature. Rispetto al modello completo richiede molte meno risorse per l'addestramento ma al contempo ha potenzialità ridimensionate.

```

class JanusImageEncoderTune(torch.nn.Module):
    def __init__(self, model_name: str, num_classes: int = 1, shape=(3, 224,
        224)):
        super().__init__()
        model_path = "deepseek-ai/"
        self.processor = janus_models.VLChatProcessor.from_pretrained(
            model_path + model_name
        )
        self.feature_encoder = AutoModelForCausalLM.from_pretrained(
            model_path + model_name, trust_remote_code=True
        )
        self.feature_encoder.gen_vision_model = None
        self.feature_encoder.gen_aligner = None
        self.feature_encoder.gen_head = None
        self.feature_encoder.gen_embed = None
        self.feature_encoder.language_model = None
        gc.collect()
        torch.cuda.empty_cache()

        dummy_image = torch.zeros((1, *shape))
        features = self.forward_features(dummy_image)
        self.fc = ChannelLinear(features.shape[1], num_classes)

```

```

    torch.nn.init.normal_(self.fc.weight.data, 0.0, 0.02)

def forward_features(self, x: Tensor):
    # image preprocess
    toPIL = transforms.ToPILImage()
    pil_images = [toPIL(img) for img in x]
    pixel_values = (
        self.processor.image_processor(pil_images, return_tensors="pt")
        .pixel_values.to(torch.bfloat16)
        .to(self.feature_encoder.device)
    )
    pixel_values = pixel_values.unsqueeze(0)
    pixel_values = pixel_values.to(torch.float32)
    bs, n = pixel_values.shape[0:2]
    images = rearrange(pixel_values, "b n c h w -> (b n) c h w")
    # [b x n, T2, D]
    images_embeds = self.feature_encoder.aligner(
        self.feature_encoder.vision_model(images)
    )
    # [b x n, T2, D] -> [b, n x T2, D]
    images_embeds = rearrange(images_embeds, "(b n) t d -> b (n t) d",
        b=bs, n=n)
    # [b, n x T2, D] -> [batch_size, T2 ,D]
    images_embeds = rearrange(images_embeds, "b (n t) d -> (b n) t d", n=n)
    features = torch.nn.AdaptiveAvgPool1d(1)(images_embeds).squeeze(-1)
    return features

def forward_head(self, x: torch.Tensor) -> torch.Tensor:
    return self.fc(x)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.forward_head(self.forward_features(x))

```

Listato 4.16: Modello basato sul feature encoder di JanusPro[17] adattato per il fine tuning.

# Capitolo 5

## Risultati e validazione dei modelli

### 5.1 Database

Il database utilizzato è composto da 390.000 immagini, suddivise in maniera equa tra immagini reali e sintetiche. In particolare, per ciascun generatore incluso nel database, sono state raccolte 5.000 immagini, garantendo così una rappresentanza bilanciata delle diverse fonti di immagini sintetiche. Un aspetto interessante riguarda la dimensione delle immagini (Figura 5.1): quelle reali risultano mediamente più grandi rispetto alle immagini sintetiche. Questa differenza è dovuta a due motivi principali. Da un lato, i generatori meno recenti, al momento del loro rilascio, erano limitati da vincoli tecnologici, sia a livello hardware che per gli algoritmi utilizzati, complicando la produzione di immagini ad alta risoluzione; dall'altro, le immagini reali, acquisite tramite dispositivi digitali di elevata qualità, sono spesso catturate in formati meno compressi e con maggiori dettagli, risultando quindi in dimensioni maggiori. Nonostante gli attuali modelli di generazione possano produrre immagini in alta risoluzione, le immagini reali tendono a essere comunque significativamente più grandi, evidenziando così una differenza qualitativa sostanziale tra le due fonti di dati.

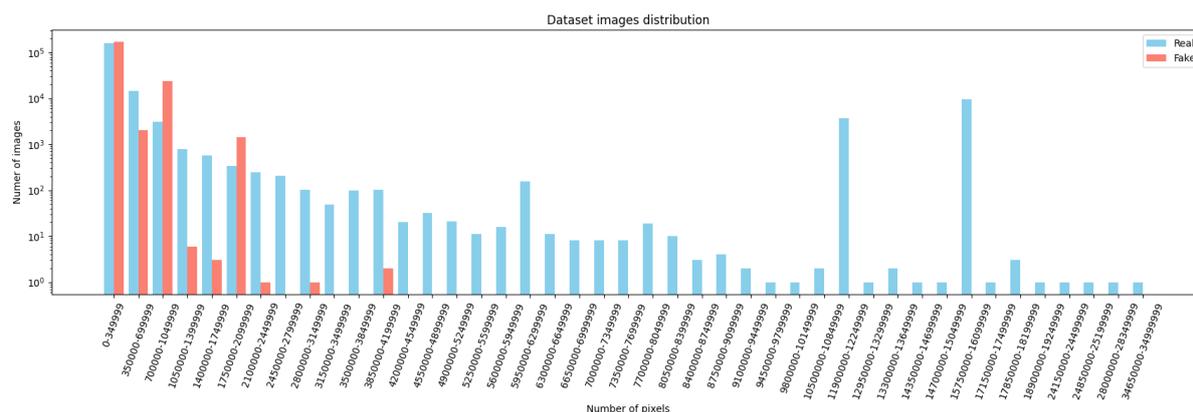


Figura 5.1: Distribuzione immagini reali e sintetiche del database per numero di pixel.

Considerando invece lo stesso database, con le immagini di dimensioni maggiori opportunamente ridimensionate a  $1080 \times 1080$  tramite il processo di crop effettuato dal benchmark, le differenze nella distribuzione delle risoluzioni si riducono (Figura 5.2); tuttavia, la maggior parte delle immagini sintetiche continua a presentare risoluzioni inferiori rispetto a quelle reali. Tali discrepanze vengono compensate mediante l'utilizzo

delle pipeline di data augmentation, che, applicando trasformazioni come crop, resize e compressione, contribuiscono a uniformare le dimensioni e a rendere il database più omogeneo per la fase di training.

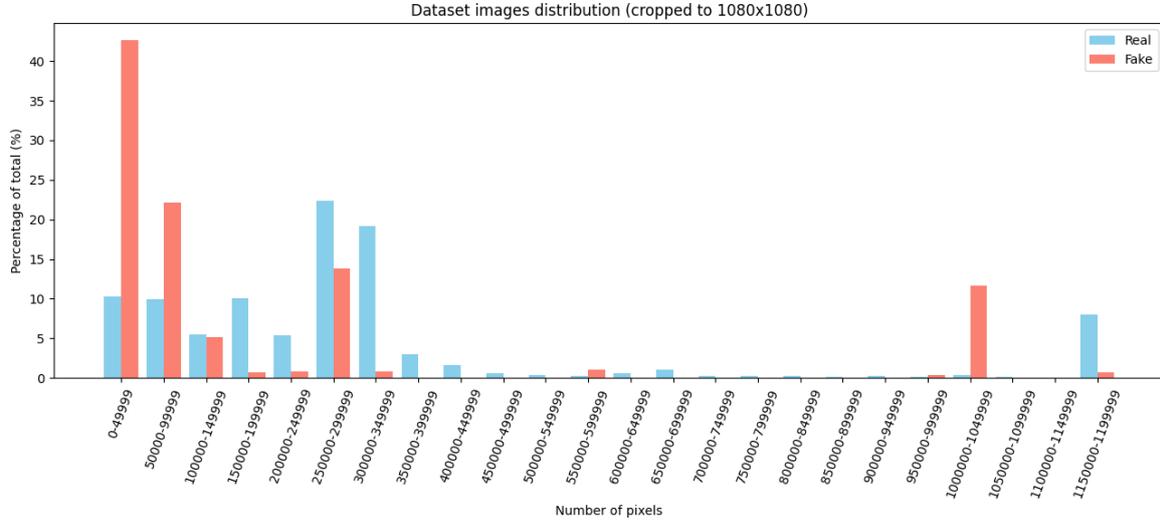


Figura 5.2: Distribuzione immagini in percentuale ritagliate a una dimensione massima di 1080x1080.

## 5.2 Metriche utilizzate

Per valutare le prestazioni dei modelli per il riconoscimento delle immagini reali e sintetiche sono state utilizzate le seguenti metriche:

- **Accuracy:** misura la percentuale di predizioni corrette rispetto al totale delle immagini. Viene calcolato come la somma di True Positive (TP) e True Negative (TN) rispetto al numero totale di elementi. Rappresenta l'efficacia complessiva del modello nel classificare correttamente entrambi i casi positivi e negativi.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision:** la precisione indica la qualità delle predizioni positive e, nel contesto del benchmark, le immagini sintetiche: quanti degli esempi etichettati dal modello come sintetici sono effettivamente tali. Una precisione elevata significa che, quando il modello identifica un'immagine come sintetica, è quasi sempre corretto, presentando un numero di False Positive (FP).

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (o True Positive Rate):** misura la capacità del modello di identificare correttamente tutti gli esempi positivi, corrispondenti alle immagini sintetiche, presenti nel database. Un valore elevato indica che il modello riesce a identificare la maggior parte degli esempi positivi, riducendo i falsi negativi (FN).

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **Average Precision:** riassume l'andamento della curva Precision-Recall, considerando tutte le possibili soglie, in un singolo valore, integrando le informazioni di precisione e recall in un'unica misura.

$$AP = \sum_{n=1}^N (\text{Recall}_n - \text{Recall}_{n-1}) \cdot \text{Precision}_n$$

- **F1 Score:** è una metrica di sintesi che combina precisione e recall in un unico valore, a differenza dell'Average Precision che considera l'andamento del modello sullo spettro delle possibili soglie, l'F1 Score fornisce una valutazione sulla soglia di classificazione utilizzata.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **AUROC (Area Under the Receiver Operating Characteristic curve):** è una metrica molto diffusa per la valutazione dei modelli di classificazione binaria. Fornisce una misura complessiva della capacità del modello di distinguere tra le classi positive e negative, considerando il comportamento del classificatore a diverse soglie. Questa metrica viene calcolata come l'area sottesa alla **Receiver Operating Characteristic curve (ROC)**[97] (Figura 5.3).

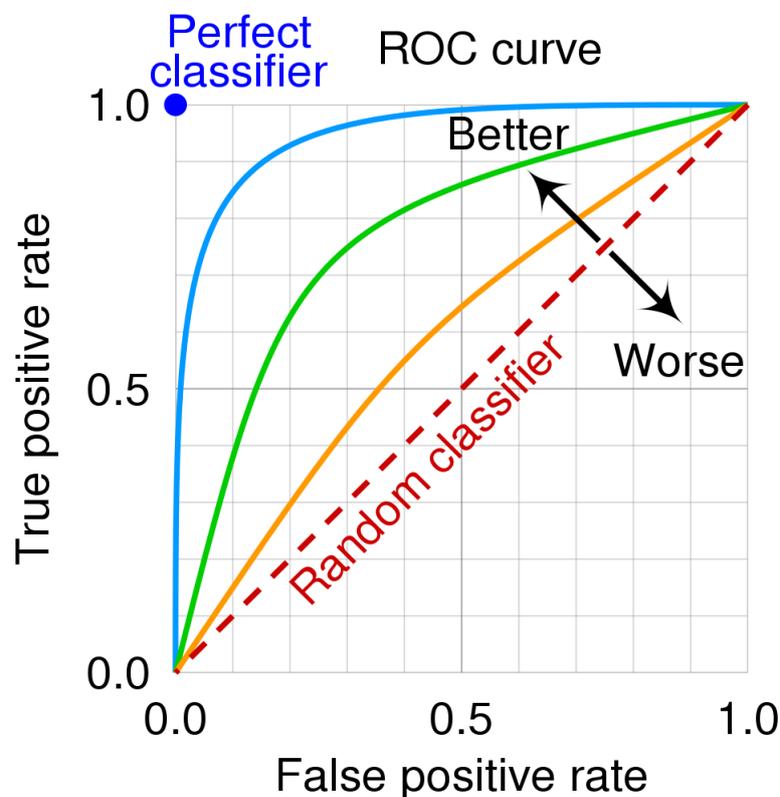


Figura 5.3: Rappresentazione della Receiver Operating Characteristic curve.

## 5.3 Validazione dei modelli

### 5.3.1 Segmentazione del validation set per Sliding Window

Per l'addestramento in modalità Sliding Window, il database viene suddiviso in sequenze ben definite, ciascuna rappresentante una "finestra" temporale. I generatori sono divisi in gruppi secondo l'ordine cronologico, a rappresentarne l'evoluzione nel tempo.

La suddivisione adottata, considerando il contenuto del database, è la seguente:

- **Sequenza 1 (dal 30/03/2017 al 24/11/2017):** CycleGAN[105], Cascaded Refinement Networks[16], ProGAN[47], StarGAN[18]
- **Sequenza 2 (dal 10/06/2018 al 12/12/2018):** SN-PatchGAN[102], BigGAN[11], IMLE[55], StyleGAN1[49]
- **Sequenza 3 (dal 18/03/2019 al 27/11/2020):** GauGAN[71], StyleGAN2[50], DDPM[39], CIPS[4]
- **Sequenza 4 (dal 17/12/2020 al 23/06/2021):** VQGAN[28], GANformer[44], ADM[25], StyleGAN3[48]
- **Sequenza 5 (dal 15/09/2021 al 10/11/2021):** LaMa[90], FaceSynthetics[98], ProjectedGAN[86], Palette[84]
- **Sequenza 6 (dal 29/11/2021 al 20/12/2021):** VQ-Diffusion[35], Denoising Diffusion GAN[96], Glide[64], Latent Diffusion[83]
- **Sequenza 7 (dal 01/02/2022 al 05/06/2022):** Midjourney[93], MAT[56], Diffusion GAN (ProjectedGAN)[96], Diffusion GAN (StyleGAN2)[96]
- **Sequenza 8 (dal 18/08/2022 al 26/04/2023):** Stable Diffusion 1.4[20], Stable Diffusion 1.5[20], Stable Diffusion 2.1[20], DeepFloyd IF[3]
- **Sequenza 9 (dal 26/07/2023 al 01/08/2024):** Stable Diffusion XL 1.0[20], DALL-E 3[68], FLUX 1 Dev[53], FLUX 1 Schnell[53]

Per ogni operazione di training eseguita sulla finestra  $F_i$  (dove  $1 \leq i \leq 9$ ), definiamo un validation set specifico per ciascuna delle seguenti configurazioni. Questo ci permette di valutare il modello da diverse prospettive.

- **Growing:** in questa configurazione, il validation set include tutte le sequenze fino alla finestra corrente  $F_i$  (con  $1 \leq i \leq 9$ ). In questo modo, la valutazione si basa sull'insieme dei dati storici accumulati fino a quel punto, fornendo una visione complessiva delle prestazioni del modello considerando l'intera storia fino alla finestra  $F_i$ .
- **Immediate Future:** in questa configurazione, il validation set è costituito esclusivamente dalla sequenza immediatamente successiva alla finestra corrente  $F_i$ , ovvero  $F_{i+1}$ . Ciò consente di valutare la capacità del modello di generalizzare sulla sequenza successiva, ossia su dati sintetici generati immediatamente dopo quelli usati per il training. Questo viene calcolato fino alla penultima finestra  $F_8$ , in modo da riservare  $F_9$  per la fase di test.

- **Growing Whole:** questa configurazione estende la modalità Growing includendo, oltre a tutte le sequenze fino alla finestra corrente  $F_i$ , anche la sequenza immediatamente successiva  $F_{i+1}$ . L'obiettivo è ottenere una valutazione più ampia, che integri sia il passato accumulato sia una proiezione sul futuro a breve termine. Nel caso dell'ultima finestra  $F_9$ , in assenza di una sequenza successiva, il validation set coincide con quello della modalità Growing.

Utilizzando queste tre configurazioni per il validation set, è possibile valutare sia le capacità del modello nel riconoscere immagini sintetiche di modelli conosciuti che di modelli non visti durante la fase di training.

### 5.3.2 Modelli utilizzati

All'interno del benchmark sono stati integrati i seguenti modelli:

- **OpenAI ResNet-50[67]:** una variante dell'architettura originale di ResNet-50 ottimizzata per il multimodal understanding, di cui si utilizza unicamente la componente visuale, 38M di parametri.
- **OpenCLIP ViT-L/14[33]:** un ViT-L/14 utilizzato come visual encoder di un modello CLIP; 303M di parametri.
- **DINOv2[80]:** l'implementazione di DINOv2 con backbone ViT-L/14; 304M di parametri.
- **DeepSeek JanusPro-feature extractor[24]:** il feature encoder per la parte di multimodal understanding del modello JanusPro-1B, adattato per estrarre le caratteristiche delle immagini senza prompt testuale; 322M di parametri.

Tutti i modelli sono pre-trained, utilizzano già dei pesi ottimali per il loro compito, ovvero l'estrazione delle feature delle immagini.

### 5.3.3 Training

#### Linear Probing

Tutti i modelli sono stati addestrati mantenendo congelati i pesi e aggiungendo un layer fully connected per la classificazione binaria. L'addestramento è stato suddiviso in nove experience, corrispondenti alle finestre temporali dei dati utilizzati, con quattro epoche per ciascuna experience.

Un'experience rappresenta una fase dell'addestramento in cui il modello apprende da un sottoinsieme dei dati, mantenendo il più possibile le conoscenze precedentemente acquisite senza dimenticarle. Nel contesto di questi addestramenti un'experience fa riferimento al singolo training del modello sulle sequenze previste in base alla configurazione scelta.

Dai risultati degli addestramenti (Figura 5.4) emerge che i modelli tendono a migliorare man mano che progrediscono le experience nelle fasi iniziali. Questo andamento è dovuto all'aumento progressivo del training e del validation set, che in ogni experience include nuovi generatori. Tuttavia, per DINOv2, ViT e RN50 si osserva un

calo generale delle prestazioni nelle esperienze dalla 6 alla 9 rispetto a quelle precedenti, probabilmente a causa dell'inclusione, sia nel training che nel validation set, di generatori basati su Diffusion Model.

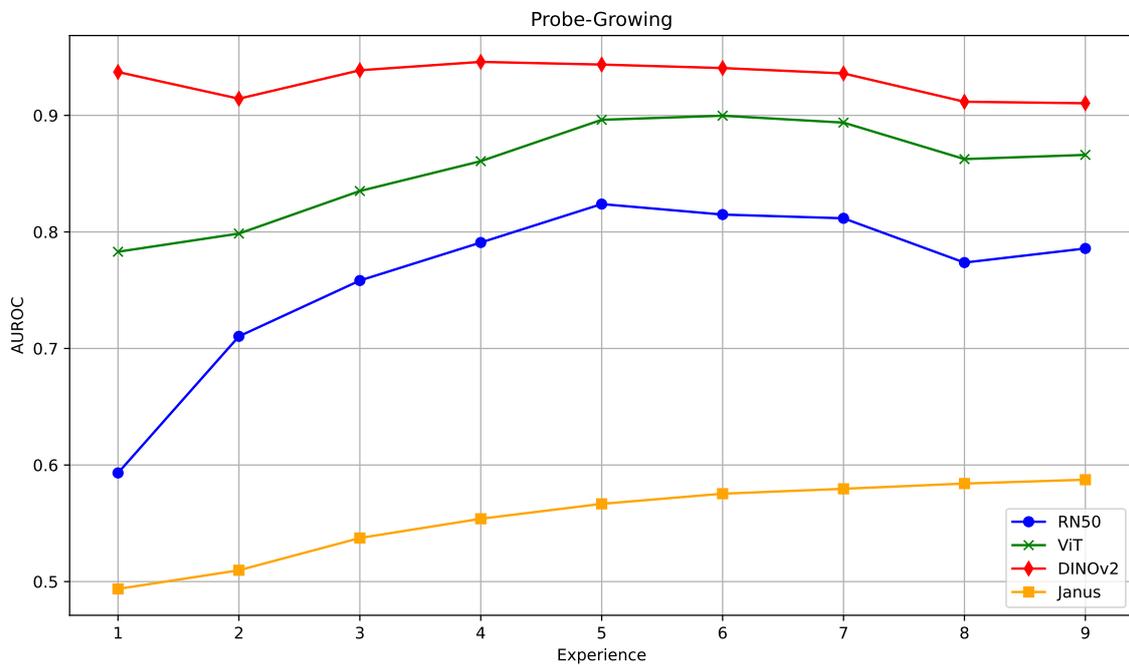


Figura 5.4: AUROC dei modelli addestrati con modalità linear probing con validation set in configurazione **Growing**.

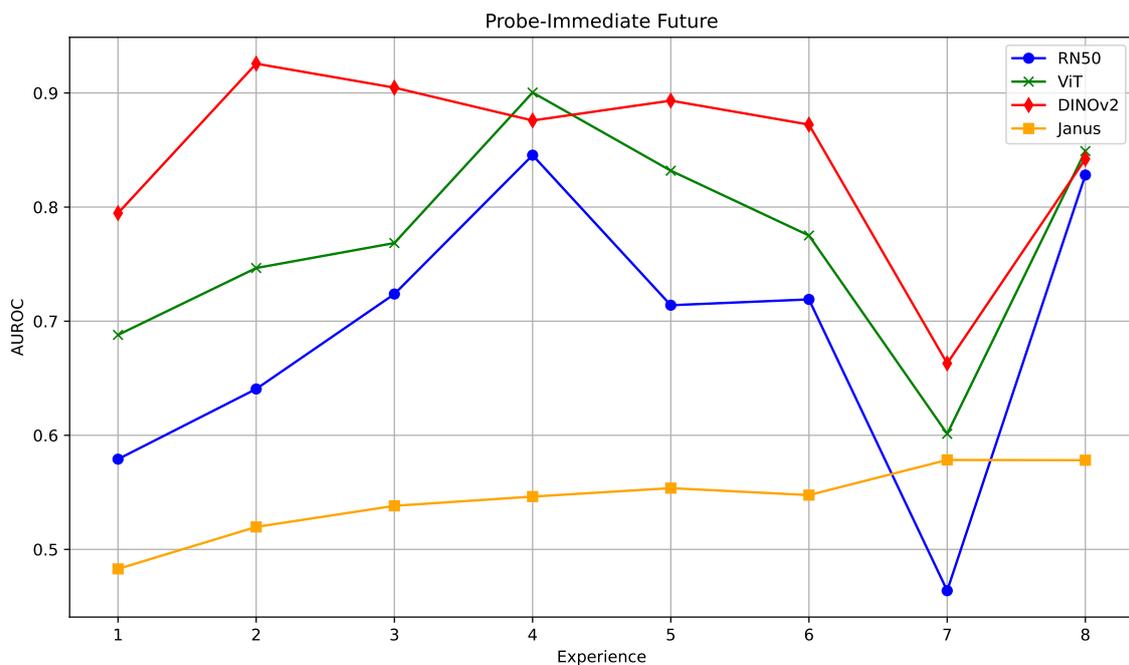


Figura 5.5: AUROC dei modelli addestrati con modalità linear probing con validation set in configurazione **Immediate Future**.

Nella valutazione Immediate Future (Figura 5.5), nelle prime experience si registra un progressivo incremento delle prestazioni, seguito da una considerevole diminuzione non appena vengono introdotti i Diffusion Model (sequenza 5). Tale degradazione risulta particolarmente evidente nella sequenza 7, in cui il validation set è composto dalla sequenza 8, prevalentemente costituita da modelli Stable Diffusion[20]. Le immagini generate da questi ultimi probabilmente presentano caratteristiche differenti rispetto a quelle delle sequenze precedenti, portando i modelli a classificarle come reali.

Il modello che si discosta maggiormente dall'andamento generale è Janus, che mostra un miglioramento nella sequenza 7 rispetto alle precedenti.

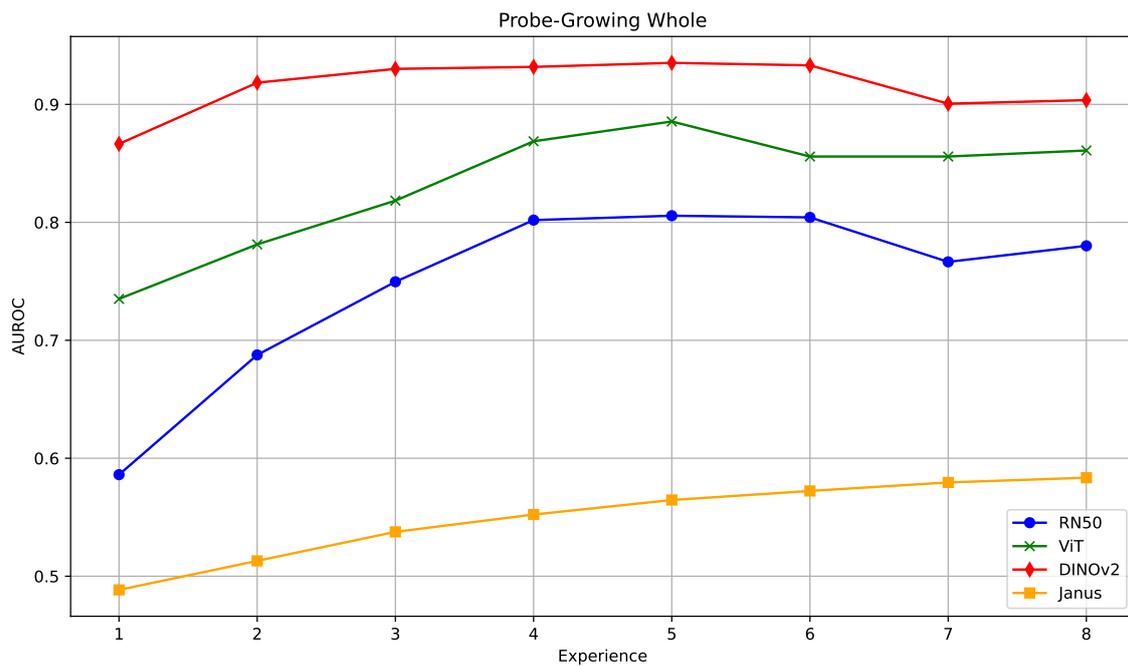


Figura 5.6: AUROC dei modelli addestrati con modalità linear probing con validation set in configurazione **Growing Whole**.

Valutando le prestazioni per Growing Whole, sia su generatori noti sia su quelli sconosciuti (Figura 5.6), si osserva una tendenza al miglioramento fino a prima dell'inclusione dei Diffusion Model. Successivamente, le prestazioni diminuiscono leggermente o rimangono stabili, con un miglioramento nell'ultima sequenza dovuto alla progressiva integrazione di ulteriori Diffusion Model nel training set. Anche in questo caso, Janus si distingue per un comportamento diverso: pur registrando prestazioni complessivamente inferiori, mostra un miglioramento costante anche dopo l'introduzione dei Diffusion Model. Tale peculiarità è probabilmente riconducibile a una combinazione di fattori, quali la complessità e l'architettura del modello. Infatti, JanusPro è stato progettato per il multimodal understanding, sfruttando un input combinato di testo e immagini; l'addestramento, tuttavia, è stato effettuato utilizzando unicamente immagini, in combinazione con il congelamento dei pesi, il che potrebbe aver influito negativamente sulle sue prestazioni.

Gli altri tre modelli addestrati presentano un andamento simile, con DINOv2 che ottiene le migliori prestazioni, seguito da ViT e RN50.

## Fine Tuning

A differenza dell'approccio Linear Probing, i pesi del modello originale vengono mantenuti addestrabili, aggiungendo sempre uno strato fully connected per la classificazione binaria in coda al modello. L'addestramento è suddiviso in nove esperienze, corrispondenti alle finestre temporali dei dati utilizzati, con quattro epoche per ciascuna experience.

I risultati ottenuti (Figura 5.7) mostrano un miglioramento generale rispetto ai training equivalenti con l'approccio Linear Probing. In particolare, si evidenzia un notevole miglioramento del modello Janus, che, potendo ottimizzare i propri pesi, raggiunge prestazioni comparabili a quelle di RN50. Nelle ultime sequenze si osserva, analogamente a quanto avviene con il Linear Probing, un lieve calo delle prestazioni dei modelli, sempre in corrispondenza dell'introduzione dei generatori di Stable Diffusion nel training e validation set.

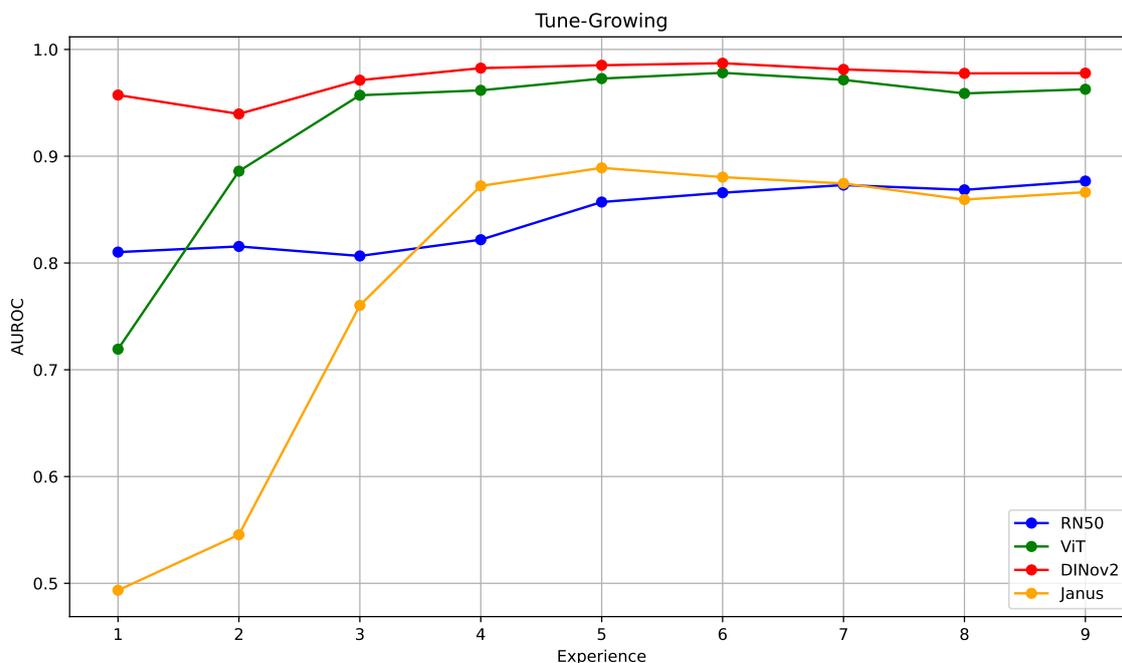


Figura 5.7: AUROC dei modelli addestrati con modalità fine tuning con validation set in configurazione **Growing**.

Nella valutazione per Immediate Future (Figura 5.8) si conferma un generale miglioramento delle prestazioni rispetto al Linear Probing. Anche in questo caso Janus si allinea agli altri modelli, mostrando un calo delle prestazioni in concomitanza con l'introduzione dei modelli basati su Stable Diffusion. Rispetto al Linear Probing, il calo delle prestazioni è meno marcato, ad eccezione di ViT. Nella sequenza successiva, dopo aver incluso i modelli di Stable Diffusion nel training set i modelli migliorano ritornando a prestazioni comparabili con la sequenza 6.

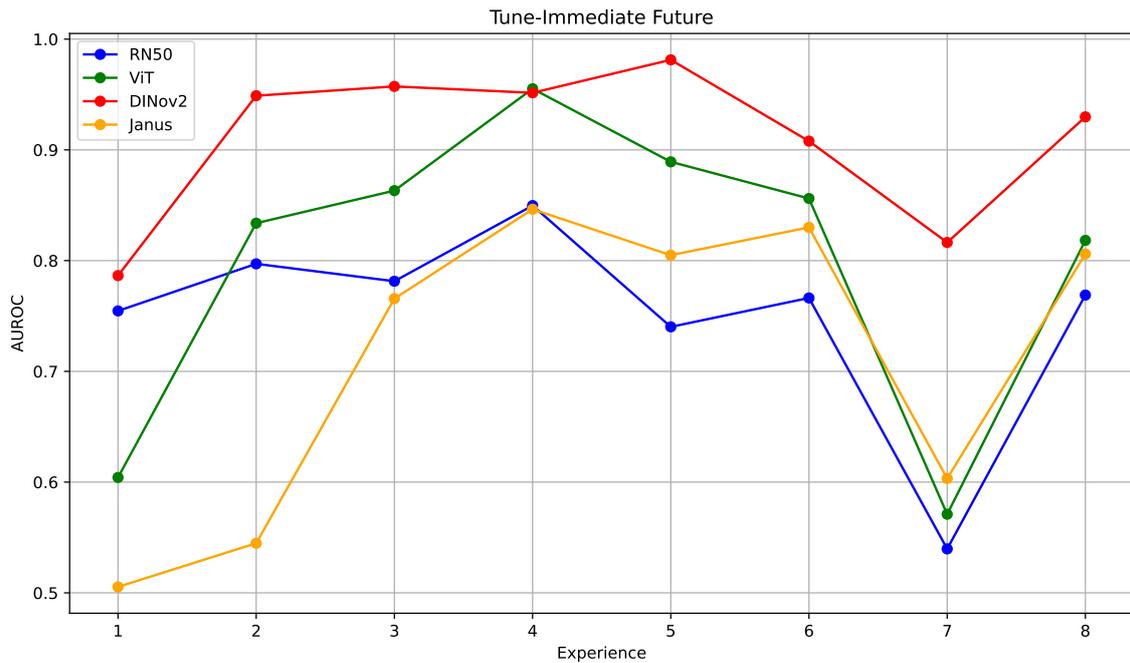


Figura 5.8: AUROC dei modelli addestrati con modalità fine tuning con validation set in configurazione **Immediate Future**.

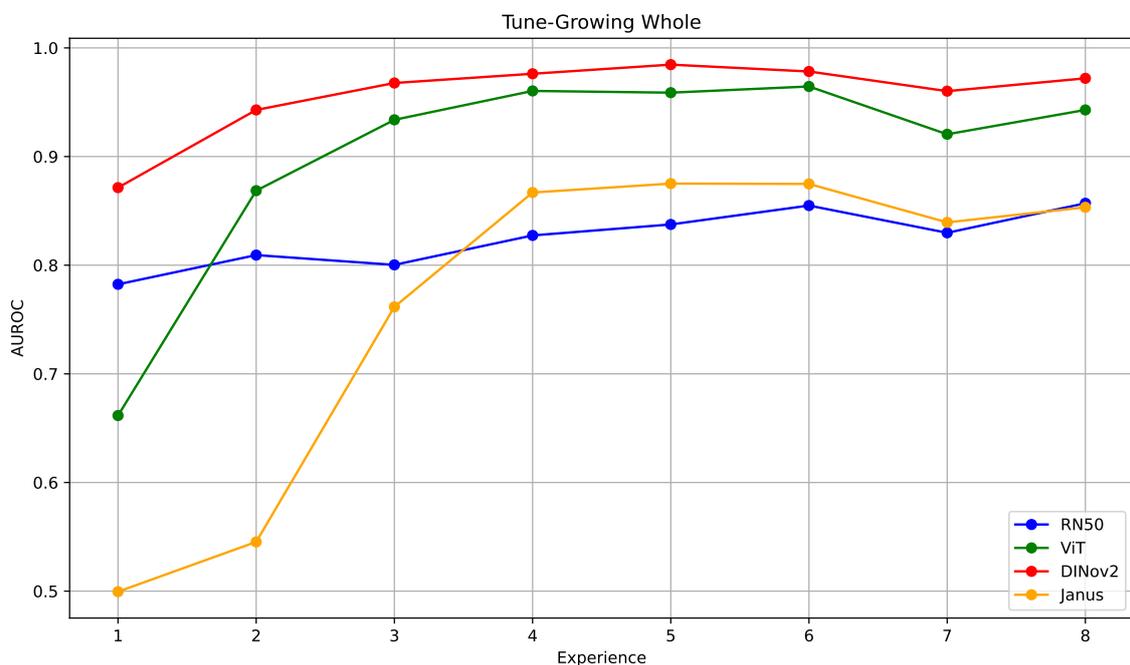


Figura 5.9: AUROC dei modelli addestrati con modalità fine tuning con validation set in configurazione **Growing Whole**.

Valutando le prestazioni complessive (Figura 5.9) si conferma il miglioramento generale, con tutti i modelli che riescono a raggiungere ottimi risultati. Confrontando i modelli DINOv2 mantiene le prestazioni migliori, seguito da ViT, e da RN50 e Janus con prestazioni molto simili. Confrontando i modelli in base al numero di pesi adde-

strabili, Janus (322M di pesi), pur conseguendo prestazioni ottime, risulta nettamente inferiore a RN50 (38M di pesi).

Valutando le prestazioni complessive (Figura 5.9) si conferma il miglioramento generale, con tutti i modelli che raggiungono ottimi risultati. Tra questi, DINOv2 mantiene le prestazioni migliori, seguito da ViT, mentre RN50 e Janus mostrano risultati molto simili. Tuttavia, se si considera il numero di pesi addestrabili, Janus (322M di pesi), pur conseguendo prestazioni eccellenti, risulta nettamente meno efficiente rispetto a RN50 (38M di pesi).

Analizzando complessivamente i risultati dei training, emerge una difficoltà comune quando nei set di training e di validation vengono inclusi i modelli di Diffusion e, in generale, i modelli più recenti. Da un lato, i modelli di Diffusion introducono una nuova categoria di immagini generate, radicalmente diversa da quelle prodotte dai modelli precedentemente utilizzati, prevalentemente basati su GAN; dall'altro, i modelli più recenti generano immagini sintetiche sempre più realistiche, complicando ulteriormente il compito di classificazione.

# Conclusioni

Il presente lavoro di tesi ha affrontato il problema sempre più pressante della manipolazione dei contenuti digitali tramite deepfake, evidenziando come la capacità dei moderni modelli di deep learning di generare immagini estremamente realistiche ponga nuove e complesse sfide nella verifica dell'autenticità delle informazioni.

L'obiettivo principale della tesi è stato quello di progettare e sviluppare un benchmark per la valutazione delle prestazioni dei sistemi di deepfake detection, con l'intento di fornire uno strumento standardizzato in grado di comparare in modo oggettivo l'efficacia dei modelli di classificazione nel riconoscimento delle immagini sintetiche.

A tal fine, sono stati analizzati in dettaglio le principali tecniche di generazione delle immagini, concentrandosi sui modelli generativi (come GAN, Autoencoder e Diffusion Model) e sulle specifiche metodologie di riconoscimento delle immagini sintetiche.

In parallelo, il lavoro ha approfondito anche lo studio dei modelli di classificazione impiegati per discriminare tra contenuti reali e sintetici. In particolare, sono state valutate soluzioni innovative che sfruttano modelli pre-addestrati come CLIP e DINOv2, specializzati nell'estrazione delle feature dalle immagini.

Il risultato raggiunto, ovvero la realizzazione di un benchmark per la valutazione delle prestazioni dei sistemi di deepfake detection, rispetta pienamente i requisiti prefissati all'inizio di questo lavoro di tesi. Il benchmark si basa su un database ampio e variegato, contenente immagini sintetiche provenienti da più di 35 modelli generativi diversi e altrettante immagini reali rappresentanti una considerevole diversità di soggetti a diverse risoluzioni. Il benchmark consente un'analisi efficace dei modelli di classificazione tramite la possibilità di configurare ogni aspetto dell'addestramento. Le metriche selezionate e le strategie per l'addestramento integrate permettono di valutare i modelli su una moltitudine di aspetti, tra tutti il saper riconoscere immagini sintetiche di modelli non presenti nei dati di training.

Il benchmark rappresenta quindi un valido strumento di confronto, con le potenzialità di fungere da base solida per la valutazione dei modelli attuali e futuri, offrendo un potenziale impatto in applicazioni reali nella lotta alla disinformazione tramite deepfake.

Uno dei principali punti di forza del benchmark risiede nella sua capacità di essere basato su un database aggiornato, che già include alcuni dei modelli generativi più recenti e attualmente allo stato dell'arte. In prospettiva futura, si prevede di ampliare ulteriormente il database includendo i nuovi modelli generativi che verranno rilasciati, in modo da mantenere il benchmark sempre al passo con le innovazioni del settore. Tale aggiornamento periodico non solo favorirà una comparazione più precisa delle prestazioni dei classificatori, ma contribuirà anche a stabilire standard di valutazione che riflettano in maniera costante l'evoluzione tecnologica.

Un'ulteriore caratteristica integrabile nel sistema di benchmarking è la realizzazione di classifiche pubbliche contenenti i risultati dei modelli. Attualmente, il confronto tra

i modelli si basa sulla valutazione effettuata su un database di riferimento comune, che garantisce una comparazione omogenea ed equa delle prestazioni. Con l'aggiornamento periodico del database saranno introdotte ulteriori classifiche, una per ogni versione del database.

Questo permetterebbe di mantenere una panoramica trasparente e aggiornata delle prestazioni dei vari modelli, permettendo di effettuare dei confronti equi tra essi nonostante l'evoluzione del database e dei modelli generativi in esso rappresentati. Inoltre, la consultazione delle classifiche storiche, basate su versioni precedenti del database, unita alla classifica relativa alla versione più recente, offrirà preziose informazioni per analizzare l'impatto dei nuovi modelli generativi.

# Ringraziamenti

Desidero ringraziare Laura che mi ha accompagnato durante questi anni in ogni progetto e per la sua compagnia in ogni giornata di studio. Ringrazio il relatore, Professor Matteo Ferrara, e il correlatore, Dottor Lorenzo Pellegrini, che mi hanno supportato nello sviluppo del progetto e nella redazione della tesi. Un ringraziamento speciale va ai miei genitori, che mi hanno sostenuto e incoraggiato nel mio percorso universitario. Grazie anche a tutti i miei amici e compagni di corso, Alessandro, Angelica, Francesco, Gianluca, Gloria, Leroy, Luis, Matteo e Sofia, per questo meraviglioso cammino insieme.



# Bibliografia

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.
- [2] Leonardo Labs AlmageLab. ELSA\_D3. [https://huggingface.co/datasets/elsaEU/ELSA\\_D3](https://huggingface.co/datasets/elsaEU/ELSA_D3).
- [3] Daria Bakshandaeva Christoph Schuhmann Ksenia Ivanova Nadiia Klokova Alex Shonenkov, Misha Konstantinov. Deepfloyd if. <https://github.com/deep-floyd/IF>.
- [4] Ivan Anokhin, Kirill Demochkin, Taras Khakhulin, Gleb Sterkin, Victor Lempitsky, and Denis Korzhenkov. Image generators with conditionally-independent pixel synthesis, 2020.
- [5] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.
- [6] Quentin Bammey. Polardiffshield. <https://github.com/qbammey/polardiffshield>.
- [7] Quentin Bammey. Synthbuster: Towards detection of diffusion model generated images. *IEEE Open Journal of Signal Processing*, 5:1–9, 2024.
- [8] Clark Barrett, Brad Boyd, Elie Bursztein, Nicholas Carlini, Brad Chen, Jihye Choi, Amrita Roy Chowdhury, Mihai Christodorescu, Anupam Datta, Soheil Feizi, Kathleen Fisher, Tatsunori Hashimoto, Dan Hendrycks, Somesh Jha, Daniel Kang, Florian Kerschbaum, Eric Mitchell, John Mitchell, Zulfikar Ramzan, Khawaja Shams, Dawn Song, Ankur Taly, and Diyi Yang. Identifying and mitigating the security risks of generative ai. *Foundations and Trends® in Privacy and Security*, 2023.
- [9] David Beniaguev. Synthetic faces high quality - text 2 image (sfhq-t2i) dataset, 2024.

- [10] Delyan Boychev and Radostin Cholakov. Imaginet: A multi-content dataset for generalizable synthetic image detection via contrastive learning, 2024.
- [11] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.
- [12] João P. Cardenuto, Jing Yang, Rafael Padilha, Renjie Wan, Daniel Moreira, Hao-liang Li, Shiqi Wang, Fernanda Andal’o, Sébastien Marcel, and Anderson Rocha. The age of synthetic realities: Challenges and opportunities. 2023.
- [13] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers, 2021.
- [14] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3d generative adversarial networks, 2022.
- [15] Baoying Chen, Jishen Zeng, Jianquan Yang, and Rui Yang. DRCT: Diffusion reconstruction contrastive training towards universal detection of diffusion generated images. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 7621–7639. PMLR, 21–27 Jul 2024.
- [16] Qifeng Chen and Vladlen Koltun. Photographic image synthesis with cascaded refinement networks, 2017.
- [17] Xiaokang Chen, Zhiyu Wu, Xingchao Liu, Zizheng Pan, Wen Liu, Zhenda Xie, Xingkai Yu, and Chong Ruan. Janus-pro: Unified multimodal understanding and generation with data and model scaling. *arXiv preprint arXiv:2501.17811*, 2025.
- [18] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation, 2018.
- [19] Eli Collins. New and better ways to create images with imagen 2, feb 2024. Accessed: 2025-02-18.
- [20] Stability AI CompVis and Runway. Stable diffusion. <https://stability.ai/stable-diffusion>, 2022.
- [21] Riccardo Corvi, Davide Cozzolino, Giada Zingarini, Giovanni Poggi, Koki Nagano, and Luisa Verdoliva. On the detection of synthetic images generated by diffusion models. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.
- [22] Davide Cozzolino, Giovanni Poggi, Riccardo Corvi, Matthias Nießner, and Luisa Verdoliva. Raising the bar of ai-generated image detection with clip, 2024.

- [23] Boris Dayma, Suraj Patil, Pedro Cuenca, Khalid Saifullah, Tanishq Abraham, Phúc Lê Khc, Luke Melas, and Ritobrata Ghosh. Dall·e mini, 7 2021.
- [24] DeepSeek. Januspro-1b model. <https://github.com/grip-unina/ClipBased-SyntheticImageDetection>, 2025.
- [25] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis, 2021.
- [26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.
- [27] David C. Epstein, Ishan Jain, Oliver Wang, and Richard Zhang. Online detection of ai-generated images, 2023.
- [28] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2021.
- [29] William Falcon. Pytorch lightning. <https://github.com/Lightning-AI/pytorch-lightning>, 2019.
- [30] Apache Software Foundation. Apache arrow: A cross-language development platform for in-memory data, 2022.
- [31] Chaoyou Fu, Peixian Chen, Yunhang Shen, Yulei Qin, Mengdan Zhang, Xu Lin, Jinrui Yang, Xiawu Zheng, Ke Li, Xing Sun, Yunsheng Wu, and Rongrong Ji. Mme: A comprehensive evaluation benchmark for multimodal large language models, 2024.
- [32] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.
- [33] grip unina. openCLIP TiT model. <https://github.com/grip-unina/ClipBased-SyntheticImageDetection>, 2024.
- [34] Jiayi Gu, Xiaojun Meng, Guansong Lu, Lu Hou, Minzhe Niu, Hang Xu, Xiaodan Liang, Wei Zhang, Xin Jiang, and Chunjing Xu. Wukong: 100 million large-scale chinese cross-modal pre-training dataset and a foundation framework, 2022.
- [35] Shuyang Gu, Dong Chen, Jianmin Bao, Fang Wen, Bo Zhang, Dongdong Chen, Lu Yuan, and Baining Guo. Vector quantized diffusion model for text-to-image synthesis, 2022.
- [36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [37] Zhenliang He, Wangmeng Zuo, Meina Kan, Shiguang Shan, and Xilin Chen. Attgan: Facial attribute editing by only changing what you want, 2018.

- [38] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.
- [39] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.
- [40] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.
- [41] Xun Huang, Arun Mallya, Ting-Chun Wang, and Ming-Yu Liu. Multimodal conditional image synthesis with product-of-experts gans, 2021.
- [42] Drew A. Hudson and Christopher D. Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering, 2019.
- [43] Drew A. Hudson and C. Lawrence Zitnick. Compositional transformers for scene generation, 2021.
- [44] Drew A. Hudson and C. Lawrence Zitnick. Generative adversarial transformers, 2022.
- [45] Adobe Inc. Adobe firefly versione 1, 2023. Accesso: 18 febbraio 2025.
- [46] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.
- [47] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2018.
- [48] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks, 2021.
- [49] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019.
- [50] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.
- [51] Sohail Ahmed Khan and Duc-Tien Dang-Nguyen. Clipping the deception: Adapting vision-language models for universal deepfake detection, 2024.
- [52] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.
- [53] Black Forest Labs. Flux. <https://github.com/black-forest-labs/flux>, 2023.
- [54] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer,

- Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.
- [55] Ke Li, Tianhao Zhang, and Jitendra Malik. Diverse image synthesis from semantic layouts via conditional imle, 2019.
- [56] Wenbo Li, Zhe Lin, Kun Zhou, Lu Qi, Yi Wang, and Jiaya Jia. Mat: Mask-aware transformer for large hole image inpainting, 2022.
- [57] Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Wayne Xin Zhao, and Ji-Rong Wen. Evaluating object hallucination in large vision-language models, 2023.
- [58] Simian Luo, Yiqin Tan, Longbo Huang, Jian Li, and Hang Zhao. Latent consistency models: Synthesizing high-resolution images with few-step inference, 2023.
- [59] Yiyang Ma, Xingchao Liu, Xiaokang Chen, Wen Liu, Chengyue Wu, Zhiyu Wu, Zizheng Pan, Zhenda Xie, Haowei Zhang, Xingkai yu, Liang Zhao, Yisong Wang, Jiaying Liu, and Chong Ruan. Janusflow: Harmonizing autoregression and rectified flow for unified multimodal understanding and generation, 2024.
- [60] Hannes Mareen, Dante Vanden Bussche, Fabrizio Guillaro, Davide Cozzolino, Glenn Van Wallendael, Peter Lambert, and Luisa Verdoliva. Comprint: Image forgery detection and localization using compression fingerprints, 2022.
- [61] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.
- [62] Yurii Nesterov. A method for solving the convex programming problem with convergence rate  $o(1/k^2)$ . *Dokl. Akad. Nauk SSSR*, 269, 1983.
- [63] Alex Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models, 2021.
- [64] Alex Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. Glide: Towards photorealistic image generation and editing with text-guided diffusion models, 2022.
- [65] Weili Nie, Nina Narodytska, and Ankit B. Patel. Relgan: Relational generative adversarial networks for text generation. In *International Conference on Learning Representations*, 2019.
- [66] Utkarsh Ojha, Yuheng Li, and Yong Jae Lee. Towards universal fake image detectors that generalize across generative models, 2024.
- [67] OpenAI. CLIP resnet-50. <https://github.com/openai/CLIP/>, 2021.
- [68] OpenAI. DALL·E. <https://openai.com/index/dall-e-3/>, 2021.

- [69] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, Mahmoud Assran, Nicolas Ballas, Wojciech Galuba, Russell Howes, Po-Yao Huang, Shang-Wen Li, Ishan Misra, Michael Rabbat, Vasu Sharma, Gabriel Synnaeve, Hu Xu, Hervé Jegou, Julien Mairal, Patrick Labatut, Armand Joulin, and Piotr Bojanowski. *Dinov2: Learning robust visual features without supervision*, 2024.
- [70] Takayuki Osakabe, Miki Tanaka, Yuma Kinoshita, and Hitoshi Kiya. *CycleGAN without checkerboard artifacts for counter-forensics of fake-image detection*. 2020.
- [71] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. *Semantic image synthesis with spatially-adaptive normalization*, 2019.
- [72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *Pytorch: An imperative style, high-performance deep learning library*, 2019.
- [73] Stanislav Pidhorskyi, Donald A. Adjeroh, and Gianfranco Doretto. *Adversarial latent autoencoders*. 2020.
- [74] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.
- [75] Orazio Pontorno, Luca Guarnera, and Sebastiano Battiato. On the exploitation of dct-traces in the generative-ai domain. In *2024 IEEE International Conference on Image Processing (ICIP)*, page 3806–3812. IEEE, October 2024.
- [76] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. *Learning transferable visual models from natural language supervision*, 2021.
- [77] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised representation learning with deep convolutional generative adversarial networks*, 2016.
- [78] Md Awsafur Rahman, Bishmoy Paul, Najibul Haque Sarker, Zaber Ibn Abdul Hakim, and Shaikh Anowarul Fattah. *Artifact: A large-scale dataset with artificial and factual images for generalizable and robust synthetic image detection*. In *2023 IEEE International Conference on Image Processing (ICIP)*, pages 2200–2204, 2023.
- [79] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. *Generating diverse high-fidelity images with vq-vae-2*, 2019.
- [80] Meta Research. *DINOv2 model*. <https://github.com/facebookresearch/dinov2>, 2023.

- [81] Jonas Ricker, Simon Damm, Thorsten Holz, and Asja Fischer. Towards the detection of diffusion model deepfakes, 2024.
- [82] Jonas Ricker, Denis Lukovnikov, and Asja Fischer. Aeroblade: Training-free detection of latent diffusion images using autoencoder reconstruction error. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9130–9140, June 2024.
- [83] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.
- [84] Chitwan Saharia, William Chan, Huiwen Chang, Chris A. Lee, Jonathan Ho, Tim Salimans, David J. Fleet, and Mohammad Norouzi. Palette: Image-to-image diffusion models, 2022.
- [85] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022.
- [86] Axel Sauer, Kashyap Chitta, Jens Müller, and Andreas Geiger. Projected gans converge faster, 2021.
- [87] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates, 2018.
- [88] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics, 2015.
- [89] Anderson A. Susskind, J. and G. E. Hinton. The toronto face dataset. technical report utml, 2010.
- [90] Roman Suvorov, Elizaveta Logacheva, Anton Mashikhin, Anastasia Remizova, Arsenii Ashukha, Aleksei Silvestrov, Naejin Kong, Harshith Goka, Kiwoong Park, and Victor Lempitsky. Resolution-robust large mask inpainting with fourier convolutions. *arXiv preprint arXiv:2109.07161*, 2021.
- [91] Diangarti Tariang, Riccardo Corvi, Davide Cozzolino, Giovanni Poggi, Koki Nagano, and Luisa Verdoliva. Synthetic image verification in the era of generative ai: What works and what isn't there yet, 2024.
- [92] Chameleon Team. Chameleon: Mixed-modal early-fusion foundation models, 05 2024.
- [93] MidJourney Team. Midjourney. <https://www.midjourney.com>, 2022.
- [94] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018.

- [95] Sheng-Yu Wang, Oliver Wang, Richard Zhang, Andrew Owens, and Alexei A Efros. Cnn-generated images are surprisingly easy to spot...for now. In *CVPR*, 2020.
- [96] Zhendong Wang, Huangjie Zheng, Pengcheng He, Weizhu Chen, and Mingyuan Zhou. Diffusion-gan: Training gans with diffusion, 2023.
- [97] Wikipedia contributors. Receiver operating characteristic — wikipedia, l'enciclopedia libera, 2025. Accesso: 5 marzo 2025.
- [98] Erroll Wood, Tadas Baltrušaitis, Charlie Hewitt, Sebastian Dziadzio, Matthew Johnson, Virginia Estellers, Thomas J. Cashman, and Jamie Shotton. Fake it till you make it: Face analysis in the wild using synthetic data alone, 2021.
- [99] Chengyue Wu, Xiaokang Chen, Zhiyu Wu, Yiyang Ma, Xingchao Liu, Zizheng Pan, Wen Liu, Zhenda Xie, Xingkai Yu, Chong Ruan, et al. Janus: Decoupling visual encoding for unified multimodal understanding and generation. *arXiv preprint arXiv:2410.13848*, 2024.
- [100] Zhisheng Xiao, Karsten Kreis, and Arash Vahdat. Tackling the generative learning trilemma with denoising diffusion gans, 2022.
- [101] Fisher Yu, Yinda Zhang, Shuran Song, Ari Seff, and Jianxiong Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.
- [102] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas Huang. Free-form image inpainting with gated convolution, 2019.
- [103] Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, Yu Su, Wenhui Chen, and Graham Neubig. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark. *arXiv preprint arXiv:2409.02813*, 2024.
- [104] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiao lei Huang, and Dimitris Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks, 2017.
- [105] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks, 2020.
- [106] Mingjian Zhu, Hanting Chen, Qiangyu Yan, Xudong Huang, Guanyu Lin, Wei Li, Zhijun Tu, Hailin Hu, Jie Hu, and Yunhe Wang. Genimage: A million-scale benchmark for detecting ai-generated image, 2023.