Università di Pisa

Dipartimento di Informatica
Corso di Laurea in Informatica

Tesi di Laurea

Tecniche di IA Generativa per l'arricchimento di dataset di immagini

Tutor accademico
Domenico Tortorella

Candidato
Giacomo Torbidoni

Tutor aziendali
Giulio Del Corso
Oscar Papini

Anno Accademico 2024/2025

Indice

1 Introduzione **1**
- 1.1 Obiettivi del tirocinio . . . 2
- 1.2 Struttura della tesi . . . 2

2 Background e stato dell'arte della Generative AI **5**
- 2.1 Deep Learning e reti neurali profonde . . . 5
  - 2.1.1 Reti neurali convoluzionali per l'elaborazione di immagini . . . 6
- 2.2 Dalle GAN ai Diffusion Models . . . 6
- 2.3 Stable Diffusion . . . 7
  - 2.3.1 Latent Diffusion Models . . . 8
  - 2.3.2 Variational Autoencoder (VAE) . . . 8
  - 2.3.3 Diffusion forward process . . . 9
  - 2.3.4 U-Net e processo di denoising . . . 10
  - 2.3.5 *Conditioning* testuale e ruolo del prompt . . . 11
- 2.4 Cosa si intende con *data augmantation* . . . 12

3 *Data augmentation* tramite intelligenza artificiale generativa **15**
- 3.1 *Inpainting* per la generazione di nuovi elementi nella scena . . . 15
- 3.2 Inpainting per l'inserimento di oggetti prestabiliti all'interno della scena 16

4 Tecnologie utilizzate **19**
- 4.1 Python come ambiente di sviluppo . . . 20
- 4.2 ComfyUI . . . 20
- 4.3 LanPaint . . . 21
  - 4.3.1 LanPaint Ksampler . . . 23
- 4.4 ACE++ Flux . . . 24
  - 4.4.1 Specializzazione del modello tramite LoRA . . . 25
  - 4.4.2 Componenti del framework . . . 25
- 4.5 Fondamenti di acquisizione video co-registrati . . . 26
  - 4.5.1 Femto Bolt Orbbec: RGB e acquisizione di profondità . . . 26
  - 4.5.2 Calcolo di misure reali in vista *depth* . . . 28
  - 4.5.3 Parametri intrinseci ed estrinseci . . . 29
  - 4.5.4 Pseudoinversa: da RGB alla *depth* . . . 29
  - 4.5.5 Utilizzo della pseudo inversa per *Physically Informed Generative AI* . . . 32

5 Implementazione **35**
5.1 Analisi della qualità e delle tempistiche di generazione al variare dei parametri di LanPaint . . . 35
5.2 Da paradigma a blocchi a Python nativo . . . 41
5.2.1 Struttura del progetto . . . 42
5.2.2 Funzioni generiche di interazione con ComfyUI . . . 43
5.2.3 LanPaint in ambiente Python . . . 45
5.2.4 ACE++ in ambiente Python . . . 48
5.3 Filtraggio automatico delle generazioni non informative . . . 52
5.4 Usare gli infrarossi e la profondità per creare vincoli fisici alla generazione 52
5.4.1 Ricavare le distanze reali dall'immagine RGB . . . 53
5.4.2 Creazione della maschera di dimensione prestabilita . . . 58

6 Sperimentazione e validazione **63**
6.1 Validazione su dataset pubblici del metodo generativo . . . 63
6.1.1 Validazione tramite IA . . . 65
6.1.2 Analisi statistica preliminare delle generazioni . . . 71
6.1.3 Analisi statistica delle immagini filtrate . . . 73
6.1.4 Analisi statistica delle immagini generate con maschera dinamica 79
6.1.5 Confronto tra generazione con maschera dinamica e scene COCO 79

7 Conclusioni **83**
7.1 Conclusioni scientifiche . . . 83
7.2 Considerazioni sul lavoro di tirocionio . . . 85

Appendici **89**
A.1 Concetti teorici di base . . . 89
A.1.1 Tensori . . . 89
A.1.2 Maschere . . . 89
A.1.3 Priors . . . 89
A.2 Tecniche di addestramento e stabilità . . . 90
A.2.1 Mode Collapse . . . 90
A.3 Prompt e configurazioni operative . . . 90
A.3.1 Struttura dei prompt testuali . . . 90

Elenco delle figure

2.1 Flusso generale di Stable Diffusion. 7
2.2 Il ruolo del VAE di Stable Diffusion. 8
2.3 Struttura dell'architettura U-Net (fonte: [2]). 10
2.4 Esempio del processo di diffusione DDIM in Stable Diffusion (immagine tratta da [1]). 11
2.5 Esempio di tecniche di *data augmentation*. 13

3.1 Inpainting tramite Stable Diffusion 16
3.2 Inpainting tramite ACE++ (vedi sezione 4.4) di un oggetto specifico all'interno della scena. 17

4.1 Interaccia grafica di ComfyUI. 21
4.2 Interaccia grafica di ComfyUI con il flusso LanPaint 22
4.3 Confronto tra le tre fasi del processo di inpainting. 23
4.4 Esempio di generazione non adeguata 27
4.5 Immagine della telecamera Femto Bolt utilizzata. 27
4.6 Immagine della vista della telecamera. (Immagine dell'autore) 28
4.7 Corrispondenza *pixel-wise* tra immagine RGB e mappa di profondità. 29
4.8 Relazione tra mappa di profondità e immagine RGB: (a) la proiezione di un punto 3D sul piano immagine è univoca quando è nota la profondità; (b) il problema inverso non ammette soluzione univoca in assenza dell'informazione di profondità. 30
4.9 La griglia che viene proiettata dal dominio della profondità al dominio dell'immagine rgb. 31
4.10 Selezionando due punti è possibile ricavare la distanza reale tra i due. 32
4.11 Esempi di proiezione di una maschera di dimensioni reali fissate (300 mm × 500 mm) a differenti distanze dalla telecamera. A parità di dimensioni fisiche, l'estensione della maschera in pixel varia in funzione della profondità del punto selezionato, riflettendo correttamente la geometria prospettica della scena. 33
4.12 Pipeline di generazione di maschere metricamente coerenti: (a) selezione e stima metrica nello spazio RGB-D; (b) applicazione della maschera con dimensioni fisiche reali. 34

5.1 Immagine di riferimento e corrispondente maschera utilizzata negli esperimenti. 37

5.2 Risultato della generazione di riferimento 37
5.3 Risultato della generazione di riferimento 38
5.4 Risultato della generazione di riferimento 38
5.5 Risultato della generazione di riferimento 39
5.6 Risultati dell'inpainting al variare del parametro Denoise. 39
5.7 Risultato ottenuto con il sampler Euler Ancestral (tempo generazione: 34.63s). 40
5.8 Risultato ottenuto con il sampler DPM++ 2M 40
5.9 Risultato ottenuto con il sampler DPM++ 2M SDE 41
5.10 Interfaccia di ComfyUI per l'esportazione in formato API 42

6.1 Confronto tra i modelli di object detection pre-addestrati in termini di tasso di riconoscimento degli oggetti inseriti tramite inpainting. . . 71
6.2 Confronto tra la *confidence* risultante per le immagini dei tre modelli scelti 72
6.3 Confronto tra il *success rate* per le immagini dei tre modelli scelti dopo il filtraggio 78
6.4 Confronto tra la *confidence* risultante per le immagini dei tre modelli scelti dopo il filtraggio 78
6.5 Tasso di successo del rilevamento sulle immagini generate mediante maschera dinamica. 80
6.6 Confidence media dei modelli sui casi di rilevamento positivo per immagini generate mediante maschera dinamica. 80
6.7 Confronto tra accordo umano–IA in due esperimenti distinti: (a) generazione con maschera dinamica (36 immagini analizzate); (b) scene tratte dal dataset COCO (117 immagini analizzate). Le percentuali riportate in ciascuna cella sono calcolate rispetto al totale delle immagini dell'esperimento considerato. I valori numerici tra parentesi indicano il numero assoluto di campioni corrispondenti a ciascuna combinazione di valutazione umano–IA. 81

Capitolo 1

Introduzione

Questa tesi tratta dell'esperienza di tirocinio svolta presso il CNR di Pisa nel periodo che va da novembre 2025 a febbraio 2026. Durante questo periodo ho avuto l'opportunità di sviluppare una pipeline di Generative AI al fine di arricchire un dataset con dati sintetici.

Negli ultimi anni, i sistemi di Deep Learning hanno assunto un ruolo centrale in numerosi ambiti. L'evoluzione delle architetture neurali ha portato allo sviluppo di modelli sempre più complessi e performanti, capaci di raggiungere livelli eccellenti in compiti quali la classificazione, segmentazione e il riconoscimento di oggetti. È importante notare però che tali risultati dipendono in modo significativo dalla quantità di dati su cui questi sistemi sono stati allenati. I modelli di riconoscimento di oggetti di ultima generazione, citiamo per esempio YOLO [7], vengono addestrati su dataset composti da decine di milioni di immagini, evidenziando come le tecniche moderne di Deep Learning e intelligenza artificiale generativa (IA generativa) siano intrinsicamente fameliche di dati. In molti contesti reali però, tali acquisizioni di dati non risultano possibili, sia per i costi elevati di acquisizione effettiva ed annotazione, sia per la natura stessa del dominio applicativo. Settori quali quelli biomedicale, industriale o scientifico sono spesso caratterizzati da una disponibilità limitata di immagini, rendendo necessari sviluppi di strategie alternative per l'acquisizione di dati. Il progetto europeo ProCAncer-I[1], per esempio, è nato per affrontare la frammentazione e la limitata disponibilità dei dataset clinici, attraverso la creazione di un'infrastruttura europea condivisa per la raccolta e l'analisi di immagini mpMRI su larga scala. Tale iniziativa evidenzia come, anche in domini ad alta rilevanza scientifica e clinica, la scarsità di dati costituisca un fattore limitante cruciale per lo sviluppo di modelli di intelligenza artificiale affidabili e generalizzabili.

La letteratura recente ha affrontato il problema della scarsità di dati proponendo diverse strategie, tra cui il *fine-tuning* di modelli pre-addestrati e l'impiego di tecniche di *data augmentation*. Numerosi studi [5, 10] evidenziano come tali approcci possano migliorare la capacità di generalizzazione dei modelli di visione artificiale in scenari caratterizzati da una limitata disponibilità di dati.

Negli ultimi anni, l'emergere dei modelli generativi ha aperto nuove prospettive nell'ambito della data augmentation, consentendo la produzione di dati sintetici

[1] https://www.procancer-i.eu

realistici e coerenti. Queste tecniche rappresentano una direzione promettente per l'estensione controllata dei dataset in contesti applicativi.

Il lavoro svolto nel presente elaborato si inserisce come sottoprogetto applicativo all'interno del progetto europeo FAITH[2], che si occupa di promuovere lo sviluppo e l'adozione di sistemi di intelligenza artificiale non solo performanti, ma anche affidabili, sicuri e trasparenti.

1.1 Obiettivi del tirocinio

In questo contesto, il lavoro svolto nel presente elaborato si concentra sulla progettazione, implementazione e validazione di un processo di data augmentation avanzata basato su modelli di intelligenza artificiale generativa. L'obiettivo non è unicamente l'ampliamento numerico di un dataset esistente, ma lo sviluppo di un metodologia controllata e riproducibile per la generazione di immagini sintetiche realistiche e coerenti da integrare in contesti visivi reali.

Il processo sviluppato è finalizzato al supporto di sistemi di visione artificiale per il riconoscimento automatico di oggetti abbandonati o rifiuti in ambienti ferroviari, quali banchine e vagoni dei treni. Il contributo del presente lavoro si colloca quindi nella fase di costruzione e arricchimento dei dati, affrontando in modo sistematico le problematiche legate alla qualità visiva, alla coerenza geometrica e alla corretta integrazione degli oggetti sintetici all'interno delle scene.

La raccolta di nuove immagini in contesti ferroviari reali non è risultata praticabile a causa di vincoli logistici, normativi e di sicurezza. Di conseguenza, il dataset iniziale a disposizione risultava numericamente limitato e caratterizzato da una ridotta varietà di scenari e configurazioni degli oggetti di interesse, rendendo necessario l'impiego di strategie alternative.

Il lavoro presentato non mira a sostituire i dati reali, ma ad arricchirli attraverso un processo automatico e scalabile, in grado di produrre grandi volumi di dati mantenendo un elevato livello di qualità.

1.2 Struttura della tesi

- Capitolo 2: In questo capitolo vengono trattate le basi dei modelli generativi orientati alle immagini e le tecnologie di IA generativa più performanti, partendo dalle GAN fino ad arrivare ai modelli allo stato dell'arte basati su diffusione e il loro funzionamento di base.

- Capitolo 3: Trattiamo in questo capitolo il significato di *data augmentation* e le tecniche oggi presenti in letteratura al fine di ampliare i dati di training.

- Capitolo 4: In questo capitolo di questo documento vengono invece elencate le tecnologie e gli strumenti utilizzati durante il tirocinio.

[2] https://faith-ec-project.eu

- Capitolo 5: in questo capitolo viene presentata l'implementazione delle tecnologie adottate per la generazione di dati sintetici, con particolare attenzione all'integrazione delle informazioni di profondità presenti nel dataset, utilizzate per migliorare il realismo delle immagini generate e la coerenza prospettica degli oggetti inseriti.

- Capitolo 6: Questo ultimo capitolo tratta di come sia stato eseguito un controllo di validazione delle immagini generate, sia mediante l'utilizzo di modelli di *object detection* sia mediante un controllo visivo manuale al fine di evidenziare e scartare gli elementi che presentavano artefatti.

- Capitolo 7, Conclusioni: In questo capitolo, che conclude il lavoro, si riflette su quanto svolto, analizzando i risultati ottenuti e gli ostacoli che si sono presentati, proponendo possibili vie per future implementazioni.

Capitolo 2

Background teorico e stato dell'arte della Generative Artificial Intelligence

In questo capitolo vengono presentate le conoscenze di base su cui si fonda il lavoro svolto. In particolare, viene descritto il percorso di analisi e selezione delle tecnologie di IA generativa più adatte al raggiungimento degli obiettivi prefissati. Tale percorso ha richiesto una valutazione comparativa dei principali modelli generativi proposti in letteratura, con particolare attenzione alle loro caratteristiche strutturali e alle possibilità di controllo offerte durante il processo di generazione. L'analisi si è concentrata non solo sulla qualità visiva dei risultati ottenibili, ma anche su aspetti fondamentali quali la stabilità dell'addestramento, la flessibilità architetturale, la capacità di integrare informazioni condizionali e il grado di controllabilità del processo generativo. Questi elementi risultano cruciali in scenari applicativi in cui è necessario guidare la generazione in modo preciso e riproducibile, come nel caso della data augmentation e dell'inserimento controllato di oggetti all'interno di scene reali.

Il capitolo introduce inoltre i principi teorici alla base dei modelli selezionati, fornendo il contesto necessario per comprendere le scelte progettuali e implementative discusse nei capitoli successivi.

2.1 Deep Learning e reti neurali profonde

Il Deep Learning rappresenta una branca dell'intelligenza artificiale basata sull'utilizzo di reti neurali artificiali profonde (*Deep Neural Networks*, DNN), ovvero modelli costituiti da più strati computazionali in grado di apprendere rappresentazioni gerarchiche dei dati. Tali modelli si ispirano al funzionamento dei neuroni biologici e si basano sull'apprendimento di pesi e bias tramite tecniche di ottimizzazione *gradient-based*, come il metodo della discesa del gradiente.

Una rete neurale artificiale è composta da neuroni organizzati in strati: uno strato di input, uno o più strati nascosti e uno strato di output. Ogni neurone esegue una combinazione lineare degli ingressi seguita dall'applicazione di una funzione di

attivazione non lineare, permettendo alla rete di modellare relazioni complesse e non lineari tra i dati.

Con il termine *Deep Learning* si fa riferimento a reti neurali caratterizzate da un numero significativo di strati. Questa architettura consente al modello di apprendere rappresentazioni sempre più astratte: dagli attributi più semplici fino a concetti ad alto livello.

2.1.1 Reti neurali convoluzionali per l'elaborazione di immagini

Nel caso delle immagini, le reti neurali completamente connesse non sono particolarmente adatte, in quanto non tengono conto della disposizione spaziale dei pixel. Per questo motivo, le architetture di deep learning per immagini utilizzano prevalentemente le *Convolutional Neural Networks* (CNN).

Le CNN rappresentano una classe specializzata di reti neurali artificiali che introducono l'operazione di convoluzione, permettendo di estrarre pattern locali come bordi, texture e forme direttamente dall'immagine.

Queste reti apprendono una rappresentazione gerarchica del contenuto visivo: i primi strati catturano caratteristiche elementari, mentre gli strati più profondi modellano strutture sempre più complesse e semanticamente significative. Questo meccanismo rappresenta la base di numerose architetture moderne impiegate nella visione artificiale, inclusi classificatori, detector di oggetti e modelli generativi.

Le reti convoluzionali rappresentano il blocco fondamentale su cui si basano architetture più avanzate, quali autoencoder, variational autoencoder e modelli di diffusione, che verranno approfonditi nelle sezioni successive.

2.2 Dalle GAN ai Diffusion Models per tecniche di inpainting

Il primo approccio oggetto di studio si basa sulle Generative Adversarial Networks (GAN) [3], che hanno rappresentato per lungo tempo uno dei principali riferimenti nello sviluppo di modelli generativi, in particolare nel dominio della sintesi di immagini. Le GAN si fondano su un meccanismo di apprendimento avversario tra due reti neurali, un generatore e un discriminatore, che competono al fine di produrre campioni sintetici indistinguibili dai dati reali. Nonostante l'elevata qualità visiva delle immagini generate e la capacità di produrre campioni altamente realistici, il paradigma GAN presenta alcune limitazioni rilevanti rispetto agli obiettivi del presente lavoro. In primo luogo, il processo di addestramento è noto per la sua instabilità, manifestata in fenomeni quali il mode collapse[1] e una forte sensibilità agli iperparametri. In secondo luogo, il controllo esplicito sul processo generativo risulta limitato, rendendo difficile ottenere risultati riproducibili e guidati in modo preciso. Infine, l'assenza di un meccanismo nativo per l'integrazione efficace di condizioni testuali

[1] Per un approfondimento sul fenomeno del Mode Collapse si veda l'appendice A.2.1.

complesse rende le GAN meno adatte a scenari in cui il prompt testuale rappresenta lo strumento principale di controllo. Per queste ragioni le reti GAN sono state scartate in favore di modelli più moderni e flessibili in grado di poter supportare un controllo diretto del risultato tramite prompt testuale.

Di conseguenza, l'attenzione dello studio si è concentrata sui modelli di diffusione [4], che rappresentano attualmente lo stato dell'arte nel campo della generazione di immagini e consentono di ottenere risultati coerenti con gli obiettivi finali del lavoro.

I modelli di diffusione si basano su un processo probabilistico che trasforma progressivamente un dato reale in rumore tramite una sequenza di passi incrementali. In particolare, il *forward process* aggiunge rumore gaussiano all'immagine originale secondo la relazione:

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I), \tag{2.1}$$

dove x_0 rappresenta il dato originale, x_t lo stato rumoroso al tempo t, ε è rumore gaussiano e $\bar{\alpha}_t = \prod_{i=1}^{t}(1 - \beta_i)$ controlla la quantità di informazione preservata nel tempo.

Questa formulazione evidenzia come ogni stato intermedio sia una combinazione lineare tra l'informazione originale e una componente di rumore, rendendo il processo di degradazione controllabile e matematicamente ben definito. Il compito del modello generativo consiste quindi nell'apprendere il processo inverso, ovvero la rimozione progressiva del rumore, al fine di ricostruire un campione realistico a partire da rumore puro.

2.3 Stable Diffusion

Il tipo di modello su cui è stato fatto uno studio approfondito è Stable Diffusion [8], un'evoluzione dei modelli di diffusione classici (si veda Figura 2.1). A differenza dei modelli di diffusione, che operano direttamente nello spazio delle immagini, Stable Diffusion adotta una formulazione latente, trasformando lo spazio in cui opera ad una dimensionalità ridotta. Questo permette una generazione con un costo computazionale ridotto rendendola più efficiente e leggera, mantenendo però un'elevata qualità dei risultati.

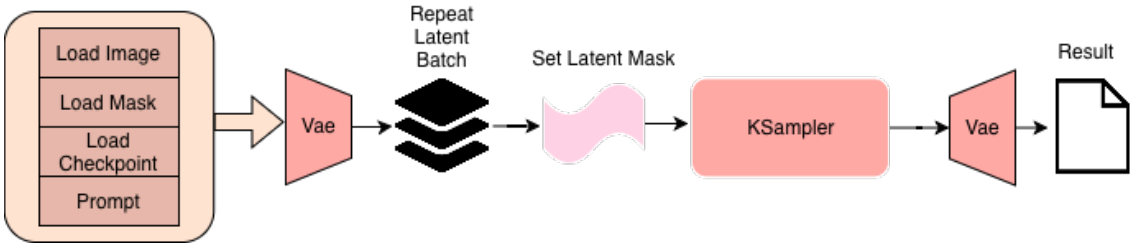


Figura 2.1: Flusso generale di Stable Diffusion.

2.3.1 Latent Diffusion Models

L'architettura generale di Stable diffusion si basa su tre componenti principali: un Variational Autoencoder (VAE), una rete U-Net e un meccanismo di prompt per guidare la generazione. In input, il sistema riceve l'immagine originale di partenza, una maschera cheidentifica la regione dell'immagine da modificare e un prompt testuale, il quale fornisce le indicazioni necessarie a guidare il modello nel processo di generazione. In uscita il modello restituisce la nuova immagine generata.

2.3.2 Variational Autoencoder (VAE)

Il Variational Autoencoder (Figura 2.2) ha il compito di trasformare l'immagine dallo spazio dei pixel a uno spazio latente più compatto e astratto. In tale spazio vengono preservate le informazioni essenziali per la generazione, riducendo la dimensionalità del dato: il processo può essere interpretato come una forma di compressione semantica dell'immagine.

A differenza di un autoencoder classico, il VAE non produce una rappresentazione latente deterministica, ma modella lo spazio latente in maniera probabilistica. Questa scelta consente di introdurre variabilità controllata nella generazione e di ottenere campioni più naturali e coerenti.

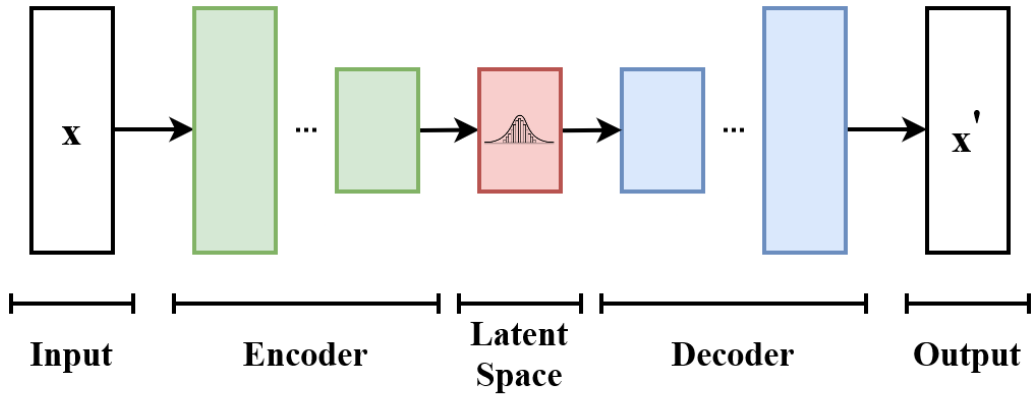


Figura 2.2: Il ruolo del VAE di Stable Diffusion.

Encoder e Decoder

Il Variational Autoencoder è costituito da due componenti principali: un encoder e un decoder. L'encoder ha il compito di trasformare l'immagine di input dallo spazio dei pixel a una rappresentazione latente probabilistica.

Durante la fase di codifica in un Variational Autoencoder, il modello non produce direttamente un singolo vettore latente deterministico, ma i parametri di una distribuzione di probabilità, tipicamente una media μ e una deviazione standard σ. Il campionamento di un vettore latente a partire da tale distribuzione introduce

tuttavia un problema nell'addestramento del modello, poiché l'operazione di campionamento non è differenziabile e impedisce l'applicazione diretta della *backpropagation* classica.

Per superare questa difficoltà viene impiegato il *reparameterization trick*, che consente di separare la componente stocastica dal processo di apprendimento. In particolare, il vettore latente viene ottenuto come

$$\mathbf{z} = \mu + \sigma \odot \boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, I),$$

dove il rumore $\boldsymbol{\varepsilon}$ è campionato da una distribuzione normale standard e la trasformazione resta differenziabile rispetto ai parametri μ e σ.

In questo modo è possibile introdurre variabilità nel vettore latente mantenendo la possibilità di addestrare il modello tramite *backpropagation*. Il vettore latente ottenuto rappresenta quindi una codifica compatta, continua e probabilistica dell'immagine di partenza.

Il decoder svolge il processo inverso, ricostruendo un'immagine nello spazio dei pixel a partire dal vettore latente campionato. Nel contesto di Stable Diffusion, il decoder del VAE viene utilizzato esclusivamente al termine del processo di diffusione, per riconvertire la rappresentazione latente finale in un'immagine visibile.

VAE vs autoencoder classico

Come già anticipato nei paragrafi precedenti, lo stato dell'arte nelle Stable Diffusion è garantito anche dal passaggio dall'autoencoder al Variational Autoencoder. Sebbene abbiano una struttura simile, caratterizzata dalla presenza di un encoder, che consente di trasformare uno spazio di pixel in uno spazio latente, e un decoder, che permette il processo inverso dopo l'intervento della U-Net, essi presentano differenze sostanziali dal punto di vista del funzionamento. L'autoencoder classico apprende una mappatura deterministica tra lo spazio dei pixel e lo spazio latente, producendo per ogni immagine di input una singola rappresentazione latente fissa. Questo comportamento risulta corretto ed adeguato per quanto riguarda compiti di compressione e ricostruzione ma risulta limitante nel contesto dei modelli generativi in quanto non garantisce continuità e campionabilità, rendendo difficile la generazione di nuovi immagini a partire da vettori latenti casuali.

Il VAE, al contrario, introducendo una componente probabilistica dello spazio latente garantisce che le rappresentazioni apprese seguano una distribuzione gaussiana. Tale proprietà permette di inizializzare il processo di generazione a partire da rumore casuale e di ottenere immagini coerenti e visivamente plausibili.

2.3.3 Diffusion forward process

Una volta proiettata l'immagine dallo spazio dei pixel allo spazio latente, il *diffusion forward process* [11] definisce una procedura di degradazione progressiva del segnale mediante l'aggiunta iterativa di rumore. A partire dalla rappresentazione latente iniziale, viene applicata una sequenza di trasformazioni che introducono rumore

gaussiano in più passi, fino a ottenere una distribuzione che approssima il rumore puro.

Dal punto di vista matematico, il *forward process* è completamente deterministico nel suo schema ed è definito da una formulazione nota a priori. Esso quindi non dipende dai dati appresi dal modello né dai parametri di addestramento, ma costituisce un processo fisso utilizzato per costruire il problema di apprendimento.

2.3.4 U-Net e processo di denoising

La U-shaped Network (U-Net) [9] 2.3 è una rete neurale convoluzionale encoder–decoder con *skip connections*, progettata per preservare informazioni locali durante processi di trasformazione e ricostruzione dei dati. Nel contesto di Stable Diffusion, il compito della U-Net è quello di individuare le componenti di rumore presenti nella rappresentazione latente, introdotte dal *diffusion forward process*, e apprendere a rimuoverle in modo progressivo. A ogni iterazione del processo di diffusione inversa, la U-Net riceve in ingresso un latente rumoroso, insieme all'informazione temporale relativa al livello di rumore, e produce una stima del rumore da sottrarre. Tale operazione viene ripetuta ciclicamente fino a ottenere una rappresentazione latente progressivamente più pulita, che può infine essere decodificata nello spazio delle immagini. È importante sottolineare che il compito effettivo di rimozione del rumore predetto dalla U-Net spetta allo *scheduler* che aggiorna progressivamente lo spazio latente fino a raggiungere uno stato del latente sufficientemente pulito da inoltrarlo al VAE Decoder a cui spetta il compito di ritrasformare lo spazio latente in pixel.

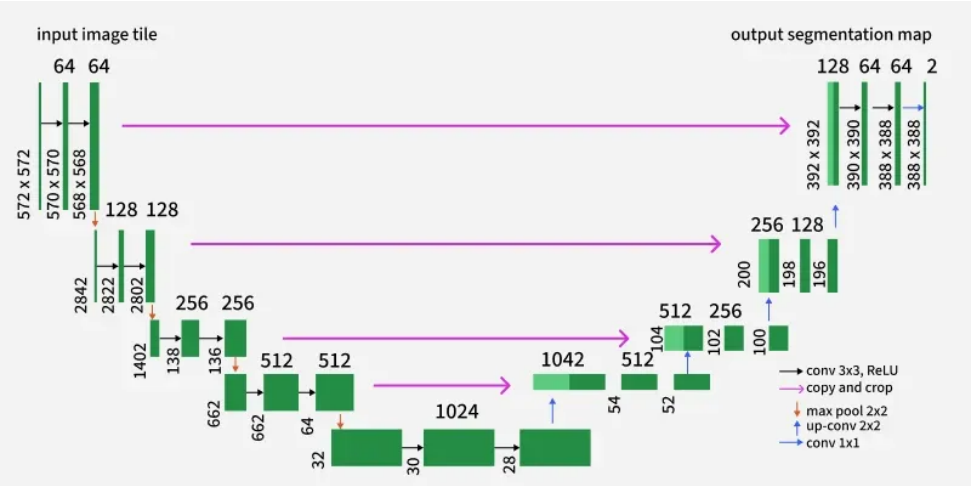


Figura 2.3: Struttura dell'architettura U-Net (fonte: [2]).

La U-Net è addestrata a stimare la componente di rumore $\varepsilon_\theta(x_t, t)$ presente nel latente rumoroso x_t al tempo t. Tale stima viene utilizzata per ottenere una versione progressivamente più pulita del latente secondo una relazione del tipo:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \varepsilon_\theta(x_t, t) \right), \tag{2.2}$$

dove α_t e $\bar{\alpha}_t$ sono coefficienti che regolano l'intensità del rumore ai diversi passi del processo di diffusione.

2.3.5 *Conditioning* testuale e ruolo del prompt

Come discusso in precedenza una caratteristica che ha convinto ad abbandonare le reti GAN per le Stable Diffusion è la possibilità di queste ultime di accettare in ingresso un prompt per poter guidare le generazioni. Il prompt rappresenta una descrizione testuale del risultato che si desidera ottenere e costituisce il principale strumento di controllo semantico del modello. Il prompt testuale, dal punto di vista operativo, viene convertito in una sequenza di tensori [2], definiti embedding, che rappresentano la codifica numerica del testo e vengono utilizzati dai moduli successivi del modello.

Tale rappresentazione viene successivamente impiegata come informazione di condizionamento durante il processo di diffusione inversa, influenzando le decisioni del modello e guidando le generazioni prodotte.

Nella figura 2.4 viene riportato un esempio dei passi che compie un modello di Stable Diffusion iterando verso una rimozione completa del rumore.

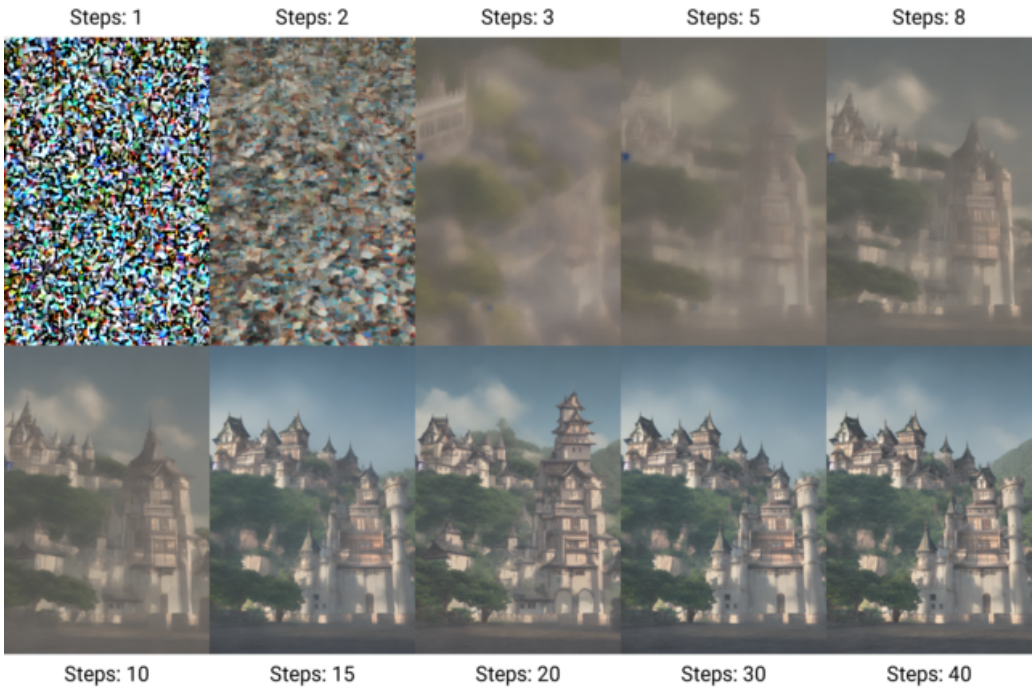


Figura 2.4: Esempio del processo di diffusione DDIM in Stable Diffusion (immagine tratta da [1]).

[2]Per un approfondimento sui tensori, si veda l'Appendice A.1.1.

2.4 Cosa si intende con *data augmantation*

Il risultato finale che il percorso di tirocinio mira ad ottenere è, come discusso nell'introduzione, l'espansione di un dataset esistente tramite tecniche di *data augmentation*. Nel suo senso più generale, la data augmentation denota metodi per integrare set di dati incompleti fornendo dati mancanti al fine di aumentare l'analizzabilità del set di dati. Ciò si manifesta nel *machine learning* generando copie modificate di dati preesistenti per aumentare le dimensioni e la diversità di un set di dati al fine di migliorare l'ottimizzazione e la generalizzazione dei modelli. In altre parole, la data augmentation punta a ridurre *l'overfitting* e migliorare la robustezza del modello.

Metodi di data augmentation

Le tecniche di *data augmentation* possono essere suddivise in diverse categorie, in base al tipo di trasformazione applicata ai dati. Nel caso delle immagini, tali metodi agiscono principalmente sulla geometria, sulle caratteristiche fotometriche, sulla qualità del segnale o sulla presenza parziale dell'informazione visiva. Nel contesto della *data augmentation* per immagini, è possibile distinguere diverse famiglie di tecniche comunemente adottate in letteratura.

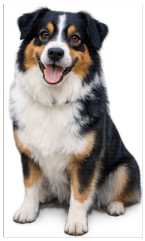
Le trasformazioni geometriche modificano la disposizione spaziale dei pixel e includono operazioni quali rotazioni, traslazioni lungo gli assi x e y, scalatura e zoom, *flip* orizzontali o verticali, trasformazioni affini (*shearing*), operazioni di ritaglio (*crop*) e *padding*, fino a deformazioni elastiche e trasformazioni prospettiche più complesse.

Le trasformazioni fotometriche agiscono invece sulle caratteristiche cromatiche dell'immagine, introducendo variazioni di luminosità, contrasto, saturazione e tonalità (*hue*), nonché correzioni gamma e operazioni di *color jitter*, senza alterare la geometria della scena.

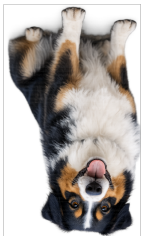
Un'ulteriore categoria è rappresentata dall'introduzione di rumore e degradazione, che mira a simulare condizioni di acquisizione non ideali mediante l'aggiunta di rumore gaussiano, rumore *salt-and-pepper* o *speckle*, l'applicazione di filtri di sfocatura (come *Gaussian* o *motion blur*) e la simulazione di artefatti di compressione, ad esempio JPEG.

Infine, le tecniche di occlusione e *masking* introducono porzioni mancanti o nascoste dell'immagine al fine di aumentare la robustezza dei modelli. Tra queste rientrano approcci quali *random erasing*, *cutout*, *hide-and-seek* e *coarse dropout*.

Alcuni dei risultati ottenuti tramite le tecniche presentate sopra sono visibili in figura 2.5.



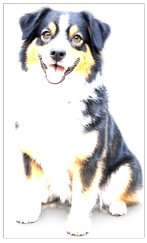
(a) Immagine originale



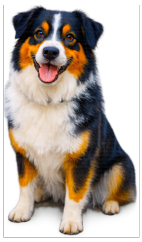
(b) Flip verticale

(c) Rotazione

(d) Blur



(e) Variazione esposizione



(f) Variazione contrasto

(g) Scala di grigi

Figura 2.5: Esempio di tecniche di *data augmentation* applicate a una singola immagine di partenza. La prima sottofigura mostra il campione originale, mentre le successive riportano diverse trasformazioni applicate.

Capitolo 3

Obiettivi e metodologia per la *data augmentation* del dataset tramite modelli di intelligenza artificiale generativa

In questo capitolo tratteremo come è possibile ampliare il dataset di partenza con la generazione di nuove immagini tramite intelligenza artificiale generativa. L'obiettivo perseguito durante lo svolgimento del tirocinio può essere sintetizzato in due fasi distinte: inserire oggetti nella scena e poter generare oggetti *ex novo* nelle scene. Un aspetto fondamentale del processo consiste nel garantire che l'oggetto generato risulti integrato in modo naturale all'interno della scena, evitando l'insorgenza di artefatti visivi o l'effetto di una sovrapposizione artificiale. L'obiettivo è ottenere immagini in cui l'elemento inserito rispetti le caratteristiche geometriche, cromatiche e luminose del contesto, risultando visivamente coerente con la scena originale. Questo aspetto si è rilevato una sfida ardua durante il tirocinio, in particolare, l'elevato volume di immagini generate rende complesso un controllo manuale dei risultati. Per questo motivo, le strategie adottate per affrontare tale problematica verranno discusse nei capitoli successivi.

3.1 *Inpainting* per la generazione di nuovi elementi nella scena

L'inpainting è una tecnica di *image editing* che consiste nel riempimento o nella ricostruzione di regioni mancanti o mascherate di un'immagine, garantendo coerenza visiva e semantica con il contesto circostante. Queste tecniche vengono ampiamente utilizzate per il restauro di immagini o la rimozione di oggetti indesiderati.

Nel contesto dei modelli generativi moderni, e in particolare dei modelli di diffusione, l'inpainting assume un ruolo più avanzato, consentendo non solo la ricostruzione di regioni mancanti, ma anche la generazione controllata di nuovi elementi all'interno di una scena. Abbiamo sfruttato questa caratteristica per la generazione

guidata di oggetti in modo da poter estendere il dataset introducendo variazioni realistiche e coerenti, preservando il contesto e le caratteristiche della scena originale. Il processo di diffusione che viene utilizzato non interviene sull'intera immagine ma solamente sulla zona delimitata dalla maschera[1] che viene passata come input al modello di diffusione insieme all'immagine originale. Quest'ultima è utilizzata dal modello di Stable Diffusion per riuscire ad estrapolare e generare correttamente il contesto visivo dell'immagine, in modo da preservare luci, ombre, texture ed eventuali sezioni da ricostruire. Nella figura 3.1 viene riportato un esempio di inpainting che rispetta le caratteristiche della scena originale.

(a) Scena originale (b) Inpainting

Figura 3.1: Inpainting tramite Stable Diffusion

3.2 Inpainting per l'inserimento di oggetti prestabiliti all'interno della scena

L'altra richiesta del progetto prevedeva, come già riportato, l'aggiunta di elementi già presenti nel dataset all'interno delle scene. Non viene più richiesta la generazione di oggetti inesistenti guidando il modello tramite prompt, ma il prompt viene utilizzato esclusivamente per descrivere l'ambiente in cui andrà a lavorare il modello e per sottomettere la richiesta. Per questo scopo è stato utilizzato un flusso diverso rispetto a quello utilizzato per la generazione di oggetti in quanto è necessario adesso fornire, oltre che l'immagine originale e la scena, anche l'oggetto di cui si desidera fare l'inpainting.

[1] Per un approfondimento sulle maschere, si veda l'Appendice A.1.2.

(a) Oggetto desiderato

(b) Scena originale

(c) Inpainting

Figura 3.2: Inpainting tramite ACE++ (vedi sezione 4.4) di un oggetto specifico all'interno della scena.

Capitolo 4

Tecnologie e framework per la *data augmentation* tramite modelli generativi

Nel presente capitolo vengono descritte le soluzioni tecnologiche adottate, che saranno approfondite nel corso delle sezioni successive. La selezione di tali tecnologie ha consentito di sperimentarne direttamente le potenzialità, valutandone in modo critico l'efficacia e l'adeguatezza rispetto agli obiettivi del progetto.

In una fase iniziale, le conoscenze relative ai modelli di Intelligenza Artificiale Generativa risultavano limitate rispetto ai requisiti tecnici richiesti. Il lavoro di studio e sperimentazione svolto nel corso del progetto ha tuttavia permesso di acquisire le competenze necessarie per utilizzare in modo consapevole ed efficace le tecnologie presentate.

Lo studio delle basi teoriche di tali modelli si è quindi svolto in parallelo alle attività di sviluppo, rappresentando una sfida particolarmente stimolante e formativa.

Le attività sperimentali sono state condotte su una workstation dedicata, equipaggiata con sistema operativo Windows 11 e dotata di una GPU NVIDIA RTX A4000 con 16 GB di memoria video, un processore Intel Core i7-12700 (12ª generazione) e 64 GB di memoria RAM. Tale configurazione ha consentito di eseguire in modo efficiente le pipeline di generazione, supportando sperimentazioni iterative e valutazioni empiriche.

Durante la fase di documentazione e ricerca inerente agli obiettivi del tirocinio, è emerso come ComfyUI[1] rappresenti una piattaforma di riferimento ricorrente in numerosi studi e implementazioni basati su modelli di diffusione e, in particolare, su Stable Diffusion. Per questo motivo, si è deciso di approfondirne il funzionamento al fine di valutarne le effettive potenzialità applicative nel contesto del lavoro svolto.

È stato osservato come molte pipeline particolarmente adatte a compiti di inpainting e data augmentation avanzata siano sviluppate direttamente su ComfyUI, grazie alla sua architettura modulare e alla possibilità di integrare modelli e compo-

[1] https://github.com/Comfy-Org/ComfyUI

nenti eterogenei in modo flessibile. Tra queste, particolare attenzione è stata rivolta alle pipeline LanPaint e ACE++, che verranno discusse nel corso del capitolo.

La selezione di tali pipeline non è avvenuta in modo puramente teorico. Dopo una fase iniziale di analisi della letteratura e delle soluzioni disponibili online, i modelli candidati sono stati sottoposti a una valutazione empirica preliminare, mediante test manuali su un insieme di immagini di prova. Tali test avevano lo scopo di verificare la capacità dei modelli di mantenere la coerenza visiva della scena, preservando contesto, illuminazione e realismo nell'inserimento o nella modifica degli oggetti.

Un fattore determinante nella scelta delle tecnologie è stato il vincolo temporale imposto dal contesto del tirocinio. Considerate le ore disponibili e la necessità di ottenere risultati concreti e valutabili in tempi ristretti, è stato privilegiato l'utilizzo di pipeline e modelli pre-addestrati che consentissero uno sviluppo e un'implementazione rapidi, riducendo al minimo la necessità di addestramenti da zero o di modifiche architetturali complesse.

L'implementazione delle tecniche proposte, descritta nelle sezioni successive, è stata pertanto realizzata utilizzando ComfyUI come ambiente di riferimento, sfruttando e adattando le pipeline disponibili per soddisfare gli obiettivi del progetto.

4.1 Python come ambiente di sviluppo

Python è stato scelto come linguaggio di riferimento per lo sviluppo degli strumenti di supporto e di automazione utilizzati nel presente lavoro, in particolare nella versione *Python 3.10*. Tale scelta è motivata sia da una buona conoscenza pregressa del linguaggio da parte dell'autore sia dalla sua ampia diffusione del contesto del *deep learning*, per la manipolazione delle immagini e per i calcoli statistici resi possibili anche grazie alle librerie disponibili. Nel contesto di questa tesi, Python non viene impiegato per la definizione o l'addestramento dei modelli generativi, che rimangono esterni e pre-addestrati, ma viene utilizzato per controllare l'intero flusso delle pipeline di generazione, costruire le maschere in modo automatizzato, variare i parametri di generazione in maniera autonoma e calcolare le statistiche e le metriche dei risultati ottenuti.

4.2 ComfyUI

ComfyUI è un software open-source basato su un paradigma di programmazione a nodi, progettato per la costruzione e l'esecuzione di pipeline per la generazione di immagini tramite modelli di diffusione. Il sistema utilizza Stable Diffusion come modello di base e consente di integrare facilmente moduli aggiuntivi, quali LoRA [2] e altri componenti avanzati, ciascuno rappresentato come un nodo all'interno di un grafo.

Dal punto di vista architetturale, ComfyUI, la cui interfaccia è presentata in figura 4.1, permette di definire in modo esplicito il flusso dei dati tra i vari componenti

[2] LoRA (cfr. Sezione 4.4.1)

della pipeline, rendendo trasparente ogni fase del processo di generazione. Ogni nodo riceve uno o più input, esegue un'operazione specifica e produce un output che viene propagato ai nodi successivi. Questo approccio consente un elevato grado di modularità, favorendo la sperimentazione e la riconfigurazione rapida delle pipeline generative.

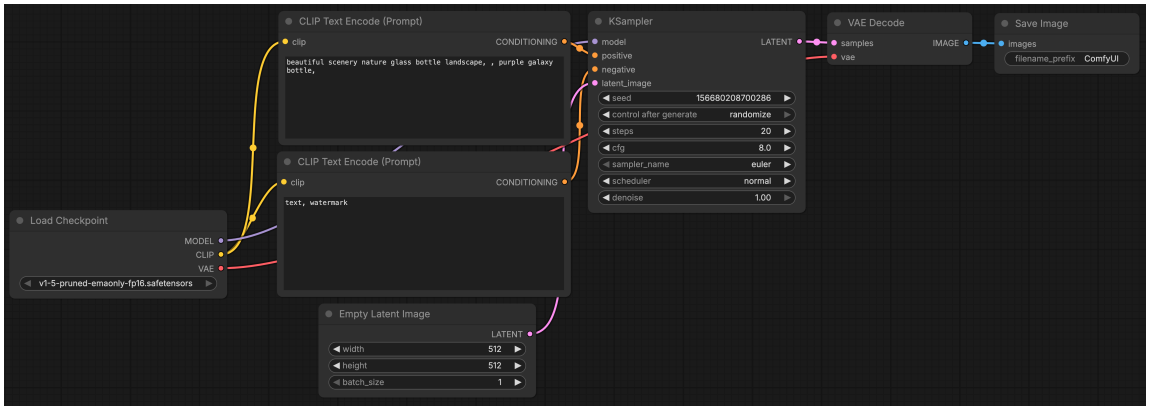


Figura 4.1: Interaccia grafica di ComfyUI.

ComfyUI non si limita a un'interfaccia grafica, ma esegue un server locale a cui è possibile inoltrare richieste e *workflow* in formato JSON tramite protocolli HTTP. Questa caratteristica permette di interagire con il sistema tramite linguaggi di programmazione, analogamente a quanto ci si aspetta da un backend API, consentendo l'automazione dei processi di generazione tramite script o chiamate di rete. Tale capacità rende ComfyUI particolarmente adatto all'integrazione in pipeline software e all'uso come backend di inferenza AI locale.

4.3 LanPaint

Uno dei flussi che più è risultato appropriato al fine del progetto è LanPaint [12]. Quest'ultimo è un approccio di inpainting basato su modelli di diffusione, progettato per consentire l'integrazione controllata di nuovi elementi all'interno di una scena tramite prompt testuale. A differenza delle tecniche di inpainting tradizionali, LanPaint non si limita alla ricostruzione di regioni mancanti, ma consente la generazione di nuovi contenuti coerenti con una descrizione testuale fornita dall'utente. Questo rende il metodo adatto a scenari di *data augmentation* avanzata. È importante sottolineare che LanPaint non definisce un modello generativo specifico, ma opera come una pipeline che sfrutta modelli di diffusione pre-addestrati. Nel contesto di questo lavoro, il modello utilizzato per la generazione delle immagini è stato scelto manualmente dall'autore, privilegiando un checkpoint orientato al fotorealismo. In particolare, è stato impiegato il modello *RealisticVisionV60B1*[3], selezionato per la sua capacità di produrre immagini ad elevato realismo visivo, con una buona coerenza

[3] https://civitai.com/models/4201/realistic-vision-v60-b1

in termini di materiali, illuminazione e dettagli fini. Tale scelta risulta particolarmente adeguata agli obiettivi del progetto, che richiedono la generazione di immagini plausibili in contesti reali come banchine e vagoni ferroviari. Un'altra opzione che è stata valutata e che ha portato risultati affini è il modello *Juggernaut XL*[4]. Gli esperimenti che seguiranno sono stati effettuati con *RealisticVisionV60B1*.

Nel corso del lavoro, LanPaint è stato utilizzato per la generazione di nuovi oggetti all'interno delle scene. Tuttavia, una limitazione intrinseca di questa pipeline riguarda l'impossibilità di fornire al modello un'immagine di riferimento di uno specifico oggetto da inserire. Tale limitazione rende necessario lo studio e l'adozione di un ulteriore approccio in grado di supportare l'inserimento guidato di oggetti a partire da un riferimento visivo.

Nella figura 4.2 viene riportato il flusso di lavoro di LanPaint.

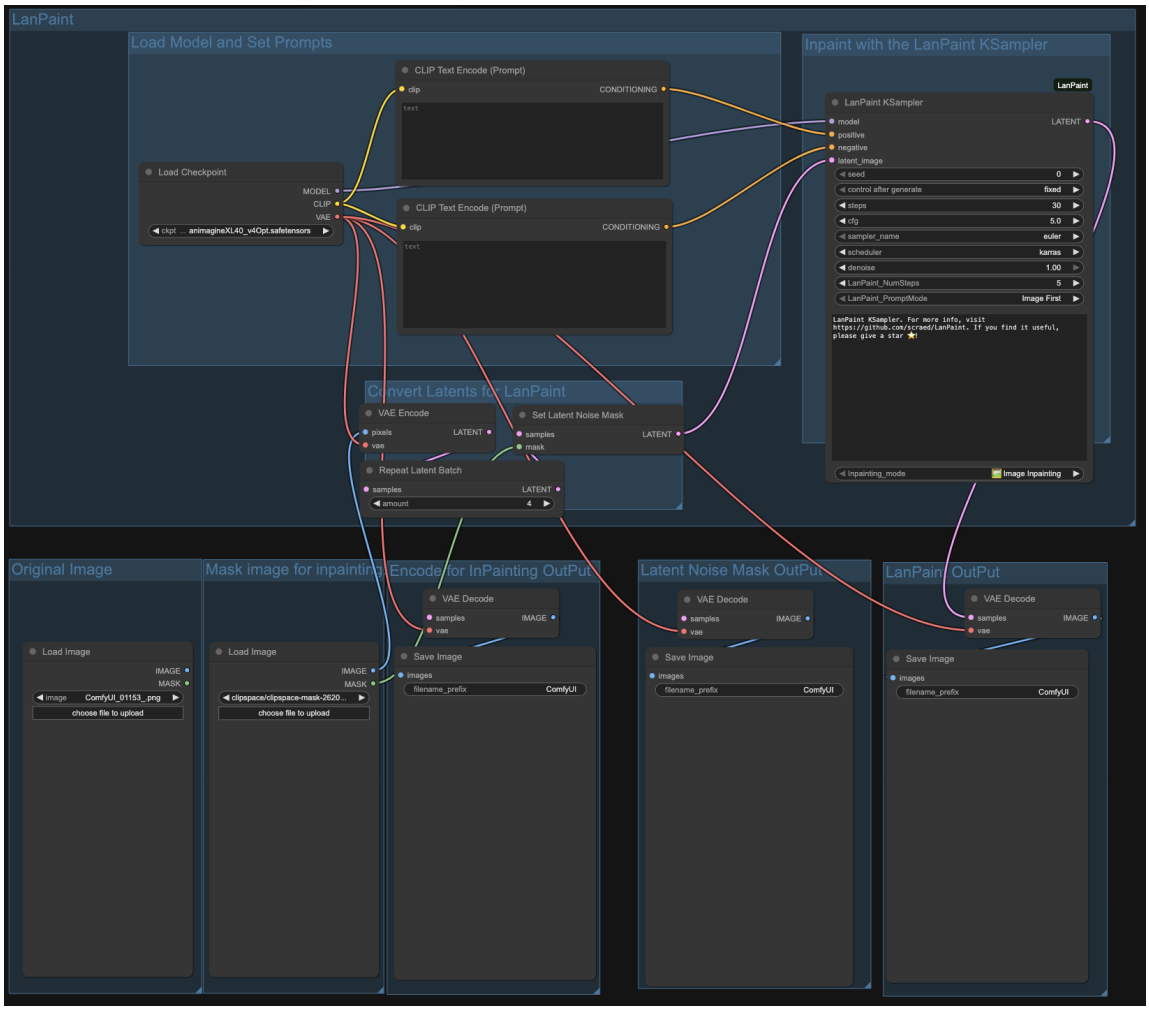


Figura 4.2: Interaccia grafica di ComfyUI con il flusso LanPaint

[4] https://civitai.com/models/133005/juggernaut-xl

4.3.1 LanPaint Ksampler

Nel contesto di LanPaint, il nodo fondamentale per la generazione delle immagini è il LanPaint KSampler, un componente basato sul concetto di *sampler* utilizzato nei modelli di diffusione per eseguire il processo di *denoising* iterativo. Nelle pipeline di ComfyUI il KSampler rappresenta il motore che effettua la transizione dallo stato latente rumoroso a una rappresentazione pulita dell'immagine mediante un numero definito di passi di diffusione.

Per la generazione il flusso necessita della scena, della maschera applicata ad essa e del prompt per guidare la generazione. Nella figura 4.3 viene proposto un esempio di generazione in cui è stata richiesta alla pipeline la generazione di una maglia utilizzando il seguente *prompt positivo*[5]:

```
generate a red t-shirt
```

(a) Originale (b) Maschera (c) Risultato

Figura 4.3: Confronto tra le tre fasi del processo di inpainting.

Parametri principali del LanPaint KSampler

I principali parametri che influenzano la generazione sono i seguenti:

- Seed: valore utilizzato per inizializzare il generatore di numeri casuali. A parità di seed e parametri, il processo di generazione risulta deterministico e riproducibile; modificando il seed si ottengono variazioni diverse del risultato generato.

[5]Per un approfondimento sui prompt positivi e negativi, si veda l'Appendice A.3.

- Steps: numero totale di passaggi di diffusione eseguiti dal *sampler*. Un valore maggiore di *Steps* consente un processo di *denoising* più graduale, potenzialmente migliorando la qualità visiva dell'immagine a fronte di un aumento del tempo di calcolo.

- CFG: *Classifier-Free Guidance scale*, parametro che regola l'intensità con cui il modello segue il prompt testuale. Valori più elevati rafforzano l'aderenza al prompt, mentre valori più bassi favoriscono una maggiore varietà e naturalezza del risultato.

- Sampler_name: algoritmo di sampling utilizzato per il processo di diffusione inversa (ad esempio *Euler*, *DDIM*, *DPM++*). La scelta del sampler influisce su stabilità, qualità visiva e velocità di convergenza del processo generativo.

- Scheduler: è utilizzato per gestire la progressione dei livelli di rumore durante il processo di diffusione. Lo scheduler definisce come i timesteps vengono distribuiti lungo il processo di denoising, influenzando il comportamento del sampler.

- Denoise: parametro che controlla l'intensità del processo di denoising applicato al latente iniziale. Valori prossimi a 1 indicano una generazione completa a partire dal rumore, mentre valori inferiori consentono una modifica parziale del contenuto esistente, risultando utili nei casi di inpainting.

- LanPaint_NumSteps: numero di iterazioni interne di ragionamento specifiche della pipeline LanPaint per ciascun passo di diffusione. Valori più elevati consentono un'elaborazione più approfondita delle informazioni contestuali, migliorando la coerenza dell'inpainting in scenari complessi.

Questi parametri consentono di bilanciare qualità, stabilità e tempi di esecuzione della pipeline, rendendo il processo adattabile sia a compiti relativamente semplici che a scenari di generazione complessi.

In particolare, la possibilità di regolare separatamente numero di passi (*Steps*) e profondità di ragionamento (*LanPaint_NumSteps*) fornisce un controllo più granulare rispetto ai sampler classici, permettendo di ottenere risultati visivamente coerenti anche in presenza di occlusioni complesse.

4.4 ACE++ Flux

Come citato in precedenza, sebbene LanPaint consenta una generazione guidata di nuovi elementi all'interno della scena tramite prompt testuale, essa non permette di fornire al modello un riferimento visivo esplicito dell'oggetto da inserire. Per superare tale limitazione è stata quindi ricercata una nuova pipeline che consentisse l'inserimento controllato di oggetti specifici all'interno della scena.

In primo luogo, è stato individuato un modello in grado di supportare tale funzionalità, mantenendo al tempo stesso la possibilità di guidare la generazione

tramite descrizione testuale. L'attenzione si è quindi concentrata sul framework ACE++ [6], che, analogamente a LanPaint, si basa su modelli di diffusione pre-addestrati. ACE++ consente di selezionare il modello più adatto allo scopo finale, permettendo di ottimizzare la qualità e la coerenza del risultato ottenuto. Il framework supporta diverse modalità di editing, tra cui l'inpainting di ritratti, la modifica dell'immagine guidata da testo e, come nel caso affrontato in questo lavoro, l'inpainting locale di oggetti all'interno della scena.

4.4.1 Specializzazione del modello tramite LoRA

Un ulteriore e rilevante vantaggio di ACE++ è l'integrazione di tecniche di *Low-Rank Adaptation* (LoRA) per la specializzazione del modello di diffusione. Le LoRA introducono correzioni a rango ridotto sui pesi del modello pre-addestrato, che rimane congelato, consentendo di adattare il sistema a compiti specifici di editing e inpainting guidato senza alterarne la struttura di base.

Dal punto di vista metodologico, questo approccio risulta particolarmente efficace in contesti in cui è richiesta un'elevata coerenza semantica e strutturale degli oggetti inseriti all'interno di scene differenti. A differenza di approcci basati esclusivamente su tecniche di guida a tempo di inferenza, come i metodi *image-to-image*, le LoRA permettono di apprendere rappresentazioni stabili e riutilizzabili di uno specifico oggetto, stile o dominio, garantendo una maggiore consistenza tra generazioni successive.

Tale strategia consente di sfruttare efficacemente i *priors*[6] generativi appresi durante la fase di addestramento del modello di base, preservandone le capacità di generalizzazione e riducendo significativamente il costo computazionale rispetto a un riaddestramento completo.

4.4.2 Componenti del framework

Il funzionamento di ACE++ si basa su tre componenti principali. Il primo è rappresentato dal modello di diffusione pre-addestrato, distribuito sotto forma di file *flux*.safetensors*. Tale file contiene il modello generativo di base, responsabile del processo di diffusione e della rimozione progressiva del rumore nello spazio latente.

Il secondo componente è costituito dai file di *Low-Rank Adaptation* (LoRA), anch'essi distribuiti nel formato *.safetensors*. Questi file consentono la specializzazione del modello verso compiti o domini specifici, permettendo di adattarne il comportamento senza modificarne direttamente i pesi originali.

Infine, il framework richiede un file di autoencoder, indicato come *ae.safetensors*, responsabile della trasformazione tra lo spazio dei pixel e lo spazio latente su cui opera il modello di diffusione.

Le tecnologie descritte in questo capitolo costituiscono la base infrastrutturale e metodologica su cui è stato costruito il sistema di data augmentation proposto. Nel capitolo successivo verrà illustrata nel dettaglio l'implementazione delle pipeline

[6] Per un approfondimento sui *priors*, si veda l'Appendice A.1.3.

descritte, insieme alle scelte sperimentali adottate e all'analisi del comportamento dei modelli al variare dei principali parametri di generazione.

4.5 Fondamenti di acquisizione video co-registrati

Quando si vanno a produrre maschere in modo manuale sull'immagine RGB è noto il problema che possono generarsi situazioni paradossali come quelle riportate in figura 4.4, ossia la generazione sproporzionata degli oggetti. Questo problema nasce dal momento che la generazione avviene in funzione della dimensione della maschera e non tiene conto delle reali dimensioni di dove viene applicata. Questo si può parzialmente mitigare sfruttando le informazioni di profondità, utilizzando la distanza dalla telecamera come criterio per la costruzione di maschere dinamiche. In questo modo, la dimensione e la posizione delle regioni di inpainting possono essere adattate in funzione della distanza degli oggetti dalla scena, garantendo una coerenza geometrica più realistica. Come discusso nell'Introduzione, il tirocinio si inserisce nel contesto del progetto europeo FAITH. In tale ambito, le acquisizioni sono effettuate mediante telecamere in grado di integrare una misura esplicita della distanza dalla scena; per questo motivo, il dataset costruito include, per ogni immagine RGB, una corrispondente mappa di profondità acquisita tramite sensore RGB-D (e.g., Orbbec Femto Bolt).

4.5.1 Femto Bolt Orbbec: RGB e acquisizione di profondità

L'acquisizione del dataset è stata effettuata mediante l'impiego di una telecamera Orbbec Femto Bolt[7], un dispositivo RGB-D che permette l'acquisizione simultanea di immagini a colori e mappe di profondità. La stima della profondità è ottenuta tramite tecnologia Time-of-Flight (ToF), misurando il tempo di ritorno di un segnale a infrarossi emesso dalla telecamera e riflesso dagli oggetti presenti nella scena. Tale caratteristica risulta fondamentale per l'analisi geometrica della scena e per l'eventuale utilizzo delle informazioni di distanza nei processi di generazione delle immagini sintetiche.

La Femto Bolt supporta differenti modalità, caratterizzate da campi visivi (*Field of View*, FoV) e risoluzioni differenti per i canali RGB e depth. Il sensore di profondità offre configurazioni sia *wide* FoV sia *narrow* FoV, con un compromesso tra ampiezza della scena osservata, risoluzione spaziale, *framerate* acquisibile e precisione della misura di profondità.

Poiché il dataset è stato acquisito con finalità di analisi su immagini statiche, e non per applicazioni *real-time*, è stato possibile privilegiare configurazioni che massimizzassero la copertura spaziale della scena e l'estensione del range di profondità, a scapito della frequenza di acquisizione. In questo modo, la scelta del FoV e della risoluzione è stata orientata a garantire una rappresentazione il più possibile completa della scena, riducendo fenomeni di clipping e aumentando la coerenza geometrica tra oggetti e sfondo.

[7] https://www.orbbec.com/products/tof-camera/femto-bolt/

(a) Maschera della zona di generazione con dimensioni fisiche non realistiche

(b) Generazione

Figura 4.4: Esempio di generazione non adeguata: in figura (a), è presente la maschera utilizzata per la generazione e in figura (b) il risultato visibilmente sproporzionato.

Figura 4.5: Immagine della telecamera Femto Bolt utilizzata.

L'adozione di tali configurazioni risulta particolarmente rilevante nel contesto del presente lavoro, in quanto la mappa di profondità viene utilizzata non solo come informazione accessoria, ma come supporto per la costruzione di maschere dinamiche e per l'imposizione di vincoli geometrici durante la generazione di immagini sintetiche. La disponibilità di misure di profondità affidabili e coerenti contribuisce quindi a migliorare il realismo prospettico degli oggetti inseriti e a ridurre incongruenze spaziali nelle immagini generate.

4.5.2 Calcolo di misure reali in vista *depth*

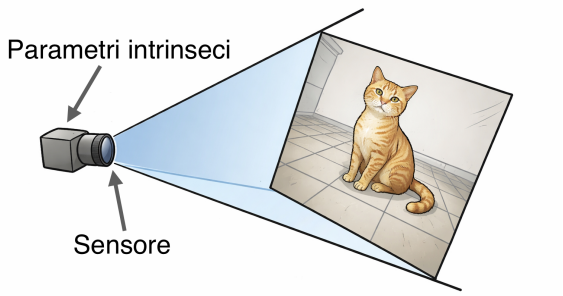


Figura 4.6: Immagine della vista della telecamera. (Immagine dell'autore)

L'acquisizione del sensore di profondità (vedi figura 4.6) attribuisce a ciascun pixel $(X_{\text{pixel}}, Y_{\text{pixel}})$ del suo piano immagine un valore di profondità D in millimetri calcolata dal sensore infrarosso. Usando questa informazione e noti i parametri intrinseci della telecamera (e.g., distanza focale, dimensione sensore, etc.), è possibile proiettare il pixel nello spazio tridimensionale reale, secondo la trasformazione:

$$(X_{\text{pixel}}, Y_{\text{pixel}}, D) \longrightarrow (X_{\text{mm}}, Y_{\text{mm}}, D)$$

dove $(X_{\text{mm}}, Y_{\text{mm}}, D)$ rappresentano le coordinate tridimensionali espresse in millimetri nel sistema di riferimento della telecamera. La componente D (profondità) rimane invariata, poiché la trasformazione agisce esclusivamente sulle coordinate planari X e Y.

In modo analogo, conoscendo i parametri intrinseci del sensore di profondità, quelli del sensore RGB e i corrispondenti parametri estrinseci (e.g., distanza fra sensori, disallineamento visuale, errore di parallasse, etc.), è possibile allineare (*co-registrare*) le informazioni di profondità all'immagine RGB. Questo consente di associare a ciascun pixel dell'immagine RGB un valore di profondità misurato, permettendo la proiezione dei punti depth direttamente nel dominio dell'immagine a colori:

$$(X^{D}_{\text{pixel}}, Y^{D}_{\text{pixel}}, D_{\text{mm}}) \longrightarrow (X^{RGB}_{\text{pixel}}, Y^{RGB}_{\text{pixel}}, D_{\text{mm}})$$

Tale corrispondenza *pixel-wise* tra immagine RGB (Figura 4.7 **a**) e mappa di profondità (Figura 4.7 **b**) costituisce la base per la costruzione di rappresentazioni RGB-

D coerenti, fondamentali per applicazioni di visione tridimensionale, ricostruzione spaziale e generazione di maschere dinamiche basate sulla distanza.

(a) Immagine RGB

(b) Mappa di profondità coregistrata sull'immagine RGB

Figura 4.7: Corrispondenza *pixel-wise* tra immagine RGB e mappa di profondità.

4.5.3 Parametri intrinseci ed estrinseci

I parametri intrinseci di una telecamera descrivono il modello di proiezione dallo spazio tridimensionale al piano immagine e dipendono esclusivamente dalle caratteristiche ottiche e geometriche del sensore. Includono, per esempio, la lunghezza focale, la posizione del centro ottico e i coefficienti di distorsione dell'obiettivo.

I parametri estrinseci, invece, definiscono la posizione e l'orientamento della telecamera rispetto a un sistema di riferimento esterno. Questi parametri consentono di mettere in relazione il sistema di riferimento della camera con quello della scena osservata o di altri sensori, come nel caso di sistemi RGB-D.

Dal punto di vista matematico, il processo di acquisizione e proiezione di una scena può essere modellato come una funzione:

$$f : \mathbb{R}^3 \longrightarrow \mathbb{R}^2,$$

che associa a ciascun punto tridimensionale della scena una coppia di coordinate nel piano immagine (Figura 4.8(a)). Per ricavare una posizione spaziale associata a un punto nel piano immagine è necessario disporre della componente di profondità. In assenza di tale informazione, il passaggio inverso dal piano immagine allo spazio tridimensionale non è univocamente definito.

4.5.4 Pseudoinversa: da RGB alla *depth*

Nel caso del sensore di profondità, la disponibilità del valore di distanza associato a ciascun pixel consente di ricostruire la posizione tridimensionale del punto nello spazio. Tale informazione permette, tramite i parametri di calibrazione intrinseci ed estrinseci dei sensori, di proiettare correttamente i punti tridimensionali sul piano dell'immagine RGB.

(a) Proiezione da mappa di profondità a immagine RGB.

(b) Ambiguità della proiezione inversa da RGB a profondità.

Figura 4.8: Relazione tra mappa di profondità e immagine RGB: (a) la proiezione di un punto 3D sul piano immagine è univoca quando è nota la profondità; (b) il problema inverso non ammette soluzione univoca in assenza dell'informazione di profondità.

Non esiste tuttavia una funzione analitica per l'inversa

$$f^{-1} : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$$

in grado di determinare in modo univoco, a partire da un singolo pixel dell'immagine RGB, la corrispondente posizione tridimensionale nello spazio. Un pixel dell'immagine RGB fornisce infatti esclusivamente informazioni bidimensionali, mentre la coordinata di profondità risulta mancante. Di conseguenza, a ciascun punto dell'immagine RGB possono corrispondere infiniti punti nello spazio tridimensionale lungo la direzione di vista della telecamera (Figura 4.8(b)).

Il problema che si presenta consiste quindi nell'identificare, a partire da una selezione effettuata nell'immagine RGB, il punto corrispondente nello spazio tridimensionale misurato dalla camera di profondità.

Per affrontare questa limitazione intrinseca, si è adottata una strategia di approssimazione basata sulla proiezione di una griglia discreta di punti dall'immagine della depth all'immagine RGB (Figura 4.10). La griglia viene definita nel dominio della depth, dove per ogni punto è disponibile una misura affidabile di profondità e quindi una coordinata tridimensionale nello spazio. Ciascun punto della griglia viene successivamente riproiettato nel sistema di riferimento dell'immagine RGB utilizzando i parametri di calibrazione tra i due sensori.

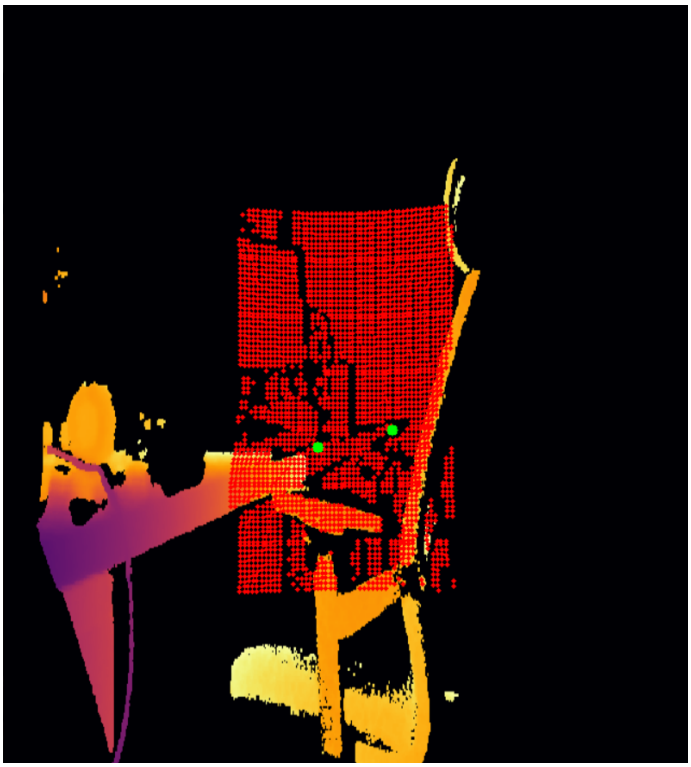


Figura 4.9: La griglia che viene proiettata dal dominio della profondità al dominio dell'immagine rgb.

In questo modo, l'immagine RGB viene arricchita da un insieme discreto di punti per i quali è nota la posizione tridimensionale. Quando l'utente seleziona un punto

sull'immagine RGB, è quindi possibile individuare il punto della griglia proiettata più vicino alla selezione effettuata. Tale punto rappresenta la migliore approssimazione disponibile della coordinata tridimensionale associata al pixel selezionato (Figura 4.9).

Figura 4.10: Selezionando due punti è possibile ricavare la distanza reale tra i due.

4.5.5 Utilizzo della pseudo inversa per *Physically Informed Generative AI*

Una volta definita una procedura in grado di approssimare, con precisione dell'ordine dei millimetri data dall'errore di stima della depth della telecamera e dall'errore di approssimazione nella griglia della pseudoinversa, la distanza associata a un punto selezionato nell'immagine RGB a partire dalle informazioni fornite dalla mappa di profondità, tale informazione può essere sfruttata per la costruzione di maschere geometricamente coerenti con la scena osservata.

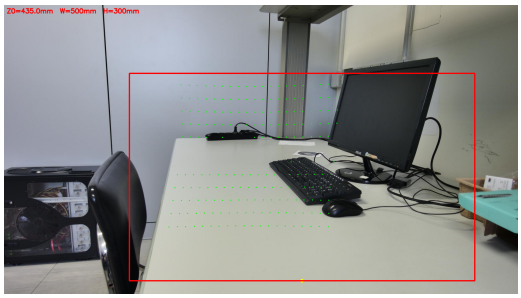
La stima della profondità consente di definire maschere le cui dimensioni sono espresse in unità metriche reali e non più in semplici coordinate pixel. Questo permette di adattare dinamicamente la forma e la scala della maschera in funzione della distanza dell'oggetto dalla telecamera, rendendo possibile la creazione di maschere specifiche per ciascun oggetto presente nella scena.

Ad esempio, come mostrato in Figura 4.12 (a), l'interazione dell'utente con l'immagine RGB consente di selezionare maschere di dimensioni reali, sfruttando le in-

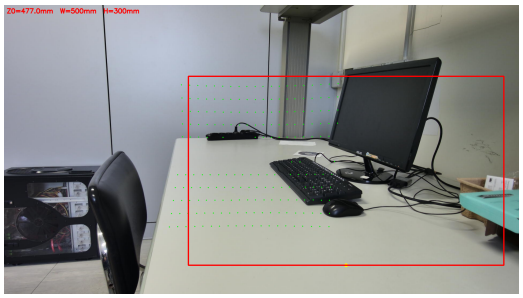
formazioni di profondità associate. A partire da tale selezione, è possibile definire una maschera rettangolare di dimensioni reali, ad esempio pari a 300 mm × 500 mm, e proiettarla correttamente nell'immagine RGB.

Il risultato finale è illustrato in Figura 4.12 (b), dove la maschera viene applicata mantenendo inalterati i valori RGB sottostanti e introducendo la trasparenza esclusivamente attraverso il canale alfa. In questo modo, l'oggetto inserito o manipolato risulta coerente non solo dal punto di vista visivo, ma anche rispetto alla scala e alla geometria tridimensionale della scena osservata.

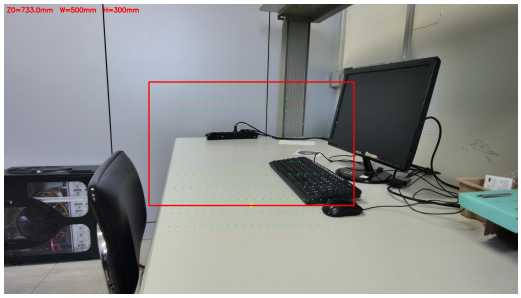
Per evidenziare il legame tra dimensioni reali e proiezione nel piano immagine, in Figura 4.11 sono riportati alcuni esempi che mostrano come una maschera di dimensioni fissate (300 mm × 500 mm) venga proiettata in modo differente in funzione della distanza dalla telecamera. A parità di dimensioni reali, l'estensione della maschera in pixel varia infatti con la profondità del punto selezionato, riflettendo correttamente la geometria prospettica della scena.



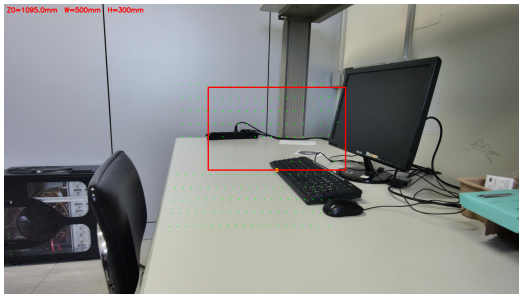
(a) Punto vicino alla telecamera



(b) Distanza intermedia



(c) Distanza maggiore



(d) Punto lontano dalla telecamera

Figura 4.11: Esempi di proiezione di una maschera di dimensioni reali fissate (300 mm × 500 mm) a differenti distanze dalla telecamera. A parità di dimensioni fisiche, l'estensione della maschera in pixel varia in funzione della profondità del punto selezionato, riflettendo correttamente la geometria prospettica della scena.

(a) Interazione sull'immagine RGB e stima delle distanze reali.

(b) Maschera finale applicata con canale alfa e proporzioni reali.

Figura 4.12: Pipeline di generazione di maschere metricamente coerenti: (a) selezione e stima metrica nello spazio RGB-D; (b) applicazione della maschera con dimensioni fisiche reali.

Capitolo 5

Implementazione delle tecniche di *data augmentation* tramite modelli generativi e inpainting

In questo capitolo vengono descritti i metodi e le scelte implementative adottate per la realizzazione delle tecniche di data augmentation basate su modelli generativi e inpainting. In particolare, il capitolo illustra il percorso sperimentale seguito per analizzare il comportamento delle pipeline selezionate e le modalità con cui tali pipeline sono state integrate in un flusso automatico.

Una prima parte del capitolo è dedicata all'analisi sperimentale dei parametri di generazione, condotta utilizzando la pipeline LanPaint come caso di studio. L'obiettivo di questa analisi è valutare l'impatto dei principali parametri sulla qualità visiva delle immagini e sulle tempistiche di esecuzione, al fine di individuare configurazioni adeguate agli scopi del progetto.

Successivamente, il capitolo descrive il passaggio dall'utilizzo delle pipeline in ambiente grafico alla loro integrazione in un ambiente Python, sfruttando le API esposte da ComfyUI. In questa fase vengono presentate le soluzioni adottate per automatizzare la generazione su larga scala, includendo sia la pipeline LanPaint sia il framework ACE++.

Nel complesso, il capitolo fornisce una visione unitaria del processo che va dalla sperimentazione sui parametri dei modelli fino all'implementazione di un sistema completo per la produzione sistematica di dati sintetici.

5.1 Analisi della qualità e delle tempistiche di generazione al variare dei parametri di LanPaint

Si può notare come LanPaint permetta un controllo fine sulla generazione agendo sui rispettivi parametri. L'implementazione ha quindi riguardato in primo luogo la fase di analisi di ciascun parametro. Di seguito vengono riportati alcuni esempi di come può variare il comportamento e la generazione del flusso variando i parametri.

Per analizzare in maniera sistematica l'effetto di questi, è stata definita una configurazione di riferimento che rimane invariata in tutti gli esperimenti, salvo dove diversamente specificato. Questo approccio garantisce che le variazioni nei risultati siano attribuibili esclusivamente al parametro in analisi, mantenendo costanti tutti gli altri fattori.

La configurazione di base include i seguenti elementi:

- Modello: RealisticVisionV60B1 (modello realistico basato su SDXL).
- Sampler: DPM++ 2M Karras.
- Numero di step: 25.
- CFG Scale: 5.0.
- Denoise strength: 0.75 (valore utilizzato da LanPaint nelle configurazioni standard).
- LanPaint NumSteps: 5.
- VAE: VAE associato al modello (automaticamente caricato dal nodo Load Checkpoint).
- Maschera: area corrispondente esclusivamente all'area da modificare.

Il prompt positivo utilizzato è il seguente:

> *"Recolor only the pencil to yellow. Keep the object exactly the same: same shape, same texture, same details, same reflections, same shadows and same lighting. Use the original image as reference. Do not modify the background. The only change must be the new color."*

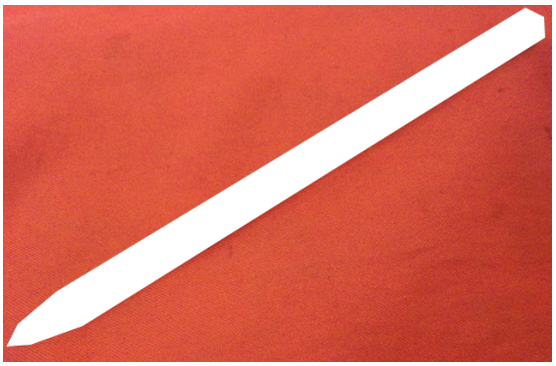
Il prompt negativo utilizzato è il seguente:

> *"blurry, deformed, distortions, incorrect details, inconsistent lighting, altered geometry, changed shape, changed reflections, changed shadows, uneven color, color bleeding, artifacts, smoothing, relighting, extra objects, invented details, new textures, incorrect materials, low quality, oversharpen."*

Come discusso in precedenza, i principali parametri della pipeline di inpainting influenzano in modo significativo il processo di generazione. Di seguito vengono riportati alcuni esempi visivi che mostrano sperimentalmente l'effetto della variazione di tali parametri sul risultato finale.

Per completezza e come punto di riferimento per i confronti successivi, in Figura 5.2 è riportato il risultato ottenuto con una generazione di base, realizzata utilizzando i parametri standard della pipeline. Questa immagine costituisce la baseline rispetto alla quale vengono valutate le successive variazioni.

(a) Immagine originale



(b) Maschera applicata

Figura 5.1: Immagine di riferimento e corrispondente maschera utilizzata negli esperimenti.

In Figura 5.5 sono mostrati i risultati ottenuti al variare del numero di step del sampler. In Figura 5.4 è invece illustrato l'effetto della variazione del parametro *CFG scale*. I risultati ottenuti modificando il parametro *LanStep* sono riportati in Figura 5.5. Infine, in Figura 5.6 è mostrato l'impatto della variazione del parametro di *denoise strength*.

Nelle figure seguenti viene illustrato l'effetto dei principali parametri della pipeline di inpainting (*steps*, *CFG scale*, *LanStep* e *denoise strength*) sul risultato finale della generazione.

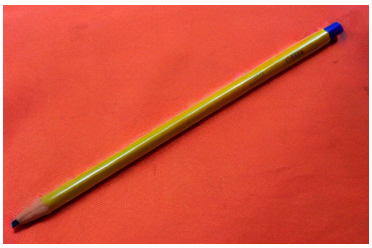
Figura 5.2: Risultato della generazione di riferimento ottenuta con i parametri standard della pipeline (tempo di generazione: 32.67 s).

Sampler Euler Ancestral

Il sampler *Euler_ancestral* 5.7 estende il metodo Euler introducendo una componente stocastica nel processo di denoising. Questa variante permette una maggiore esplorazione dello spazio latente, producendo risultati leggermente più creativi rispetto al

(a) Steps = 5
Tempo = 6.37s

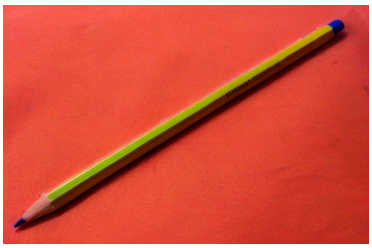
(b) Steps = 10
Tempo = 13.32s



(c) Steps = 15
Tempo = 21.32s

(d) Steps = 20
Tempo = 27.79s

(e) Steps = 25
Tempo = 34.84s



(f) Steps = 30
Tempo = 42.64s

Figura 5.3: Risultati dell'inpainting al variare del numero di step del sampler. Le sottofigure (a–f) mostrano l'effetto del parametro steps sulla qualità dell'immagine generata.

(a) CFG = 2
Tempo = 34.50s

(b) CFG = 4
Tempo = 35.36s

(c) CFG = 6
Tempo = 35.71s

(d) CFG = 8
Tempo = 35.98s

(e) CFG = 10
Tempo = 36.20s

Figura 5.4: Risultati dell'inpainting al variare del parametro CFG Scale. Le sottofigure (a–e) mostrano la sensibilità del modello al peso del prompt testuale.

(a) LanStep = 5
Tempo = 35.76s

(b) LanStep =10
Tempo = 65.88s

(c) LanStep = 15
Tempo = 96.98s

(d) LanStep = 20
Tempo = 129.96s

(e) LanStep = 25
Tempo = 163.87s

(f) LanStep = 30
Tempo = 194.70s

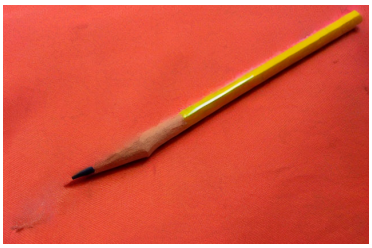
Figura 5.5: Risultati dell'inpainting al variare del parametro LanPaint_numsteps.

(a) Denoise = 0.10
Tempo = 34.71s

(b) Denoise = 0.30
Tempo = 35.50

(c) Denoise = 0.50
Tempo = 34.91s

(d) Denoise = 0.70
Tempo = 35.94s



(e) Denoise = 0.90
Tempo = 36.16s

(f) Denoise = 1.00
Tempo = 36.39s

Figura 5.6: Risultati dell'inpainting al variare del parametro Denoise.

metodo deterministico. In compiti di recolor mantiene una buona coerenza strutturale, ma può introdurre leggere variazioni cromatiche o texture più marcate. Risulta generalmente stabile e solo marginalmente più lento del sampler Euler standard.

Figura 5.7: Risultato ottenuto con il sampler Euler Ancestral (tempo generazione: 34.63s).

Sampler DPM++ 2M

Il sampler *DPM++ 2M* (Figura 5.8) rappresenta uno dei metodi più utilizzati nei modelli di diffusione moderni grazie all'ottimo compromesso tra stabilità, nitidezza e fedeltà dei dettagli. È un metodo deterministico a due step multipli che permette un controllo molto preciso sulla rimozione del rumore, producendo risultati coerenti e ben definiti. Nel contesto del recolor mantiene perfettamente la struttura dell'oggetto e rispetta le indicazioni del prompt in modo più affidabile rispetto ai metodi Euler. Risulta tuttavia leggermente più lento a parità di step.

Figura 5.8: Risultato ottenuto con il sampler DPM++ 2M (tempo generazione: 34.30s).

Sampler DPM++ 2M SDE

Il sampler *DPM++ 2M SDE* (Figura 5.9) è una variante stocastica del metodo DPM++ 2M, basata sull'integrazione stocastica delle equazioni differenziali (SDE).

Questa formulazione incrementa la stabilità e riduce la tendenza alla perdita di coerenza cromatica tipica dei sampler puramente deterministici, particolarmente nei compiti di inpainting su regioni complesse. I risultati ottenuti sono spesso più morbidi e naturali, con un miglior equilibrio tra il nuovo contenuto generato e il contesto originale. È tuttavia uno dei metodi più lenti tra quelli testati.

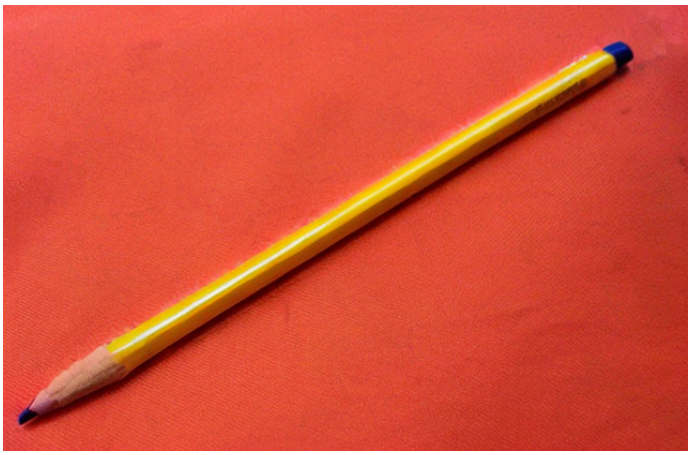


Figura 5.9: Risultato ottenuto con il sampler DPM++ 2M SDE (tempo generazione: 35.78s).

5.2 Da paradigma a blocchi a Python nativo

Nell'ambito dell'implementazione delle pipeline LanPaint e ACE++, si rende necessario integrare i flussi di generazione all'interno di uno script Python, con l'obiettivo di automatizzare e scalare il processo di data augmentation. L'implementazione mira a orchestrare in modo programmabile le diverse fasi della pipeline, includendo il caricamento delle immagini di scena e degli oggetti di riferimento, la gestione delle maschere, la configurazione dei parametri di generazione e il salvataggio dei risultati.

Per realizzare tale integrazione, sono possibili due strategie principali. La prima consiste nell'effettuare un'analisi diretta del codice sorgente di ComfyUI, che è open-source e distribuito sotto licenza MIT, con l'obiettivo di replicare o richiamare direttamente il comportamento dei singoli nodi tramite ingegneria inversa. Questo approccio consentirebbe un controllo molto fine sull'esecuzione dei workflow, ma richiede una comprensione approfondita dell'architettura interna del framework e dei meccanismi di esecuzione dei nodi, risultando particolarmente oneroso in termini di tempo e complessità.

La seconda strategia prevede invece l'utilizzo delle API esposte da ComfyUI e del relativo backend, che permettono di interagire con il sistema tramite chiamate HTTP. In questo scenario, i workflow vengono definiti come strutture JSON e inviati al server di ComfyUI, che si occupa dell'esecuzione della pipeline e della gestione delle dipendenze tra i nodi. I risultati della generazione vengono quindi resi disponibili tramite endpoint.

Considerate le esigenze di flessibilità, manutenibilità e rapidità di sviluppo, l'implementazione adotta la seconda strategia. L'interazione con ComfyUI tramite API consente infatti di riutilizzare direttamente le pipeline già validate all'interno dell'interfaccia grafica, mantenendo una perfetta corrispondenza tra workflow visuale e workflow programmato. Inoltre, questo approccio permette di separare la logica di controllo, implementata in Python, dal motore di generazione vero e proprio, favorendo una struttura modulare e facilmente estendibile.

Attraverso l'uso delle API di ComfyUI, le pipeline LanPaint e ACE++ vengono quindi integrate all'interno di script Python che gestiscono in modo automatico l'esecuzione ripetuta delle generazioni, la variazione dei parametri e dei seed casuali, e l'organizzazione dei risultati, rendendo possibile la produzione di grandi volumi di dati sintetici.

5.2.1 Struttura del progetto

Il primo passo al fine di riuscire ad utilizzare i modelli tramite API è quello di esportare il workflow su cui abbiamo lavorato in formato JSON, tramite Export(API) (Figura 5.10).

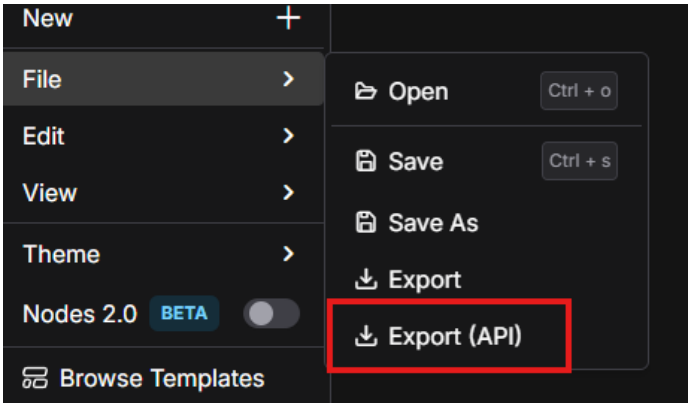


Figura 5.10: Interfaccia di ComfyUI per l'esportazione in formato API

Dal punto di vista strutturale i due workflow hanno elementi in comune, indipendente dalla pipeline specifica. Ogni workflow è identificato da un insieme di nodi identificati da un indice univoco, ciascun nodo rappresenta un'operazione elementare come il caricamento del modello in memoria, il caricamento dell'immagine e della maschera o i nodi di encoding. Ogni nodo è caratterizzato da:

- un tipo di operazione;
- un insieme di input parametrizzati;
- uno o più output utilizzabili dai nodi successivi.

```
"2": {
  "inputs": {
    "unet_name": "flux1-fill-dev.safetensors",
    "weight_dtype": "fp8_e4m3fn"
  },
  "class_type": "UNETLoader",
  "_meta": {
    "title": "Load Diffusion Model"
  }
}
```

Codice 5.1: Nodo di caricamento del modello di diffusione (U-Net).

5.2.2 Funzioni generiche di interazione con ComfyUI

Le funzioni descritte in seguito sono comuni ai due flussi e rappresentano le funzionalità di base di ComfyUI tramite API.

Upload Image

Il codice 5.2 rappresenta la funzione utilizzata per caricare immagini di input nel backend di ComfyUI tramite chiamate HTTP. La funzione verifica preliminarmente l'esistenza del file, prepara una richiesta multipart/form-data e invia l'immagine al server, che restituisce un riferimento JSON utilizzabile nei nodi successivi del workflow. Questo meccanismo consente di separare la gestione delle risorse di input dalla definizione della pipeline generativa.

```
def upload_image(path: str) -> dict:
    if not os.path.isfile(path):
        raise FileNotFoundError(f"File not found: {path}")

    with open(path, "rb") as f:
        files = {
            "image": (
                os.path.basename(path),
                f,
                "application/octet-stream"
            )
        }
        data = {
            "overwrite": "true",
            "type": "input"
        }

        r = requests.post(
            f"{COMFY}/upload/image",
            files=files,
```

```
            data=data,
            timeout=120
        )

    r.raise_for_status()
    return r.json()
```

Codice 5.2: Funzione Python per il caricamento di un'immagine nel backend di ComfyUI.

Invio del workflow a ComfyUI

Il codice 5.3 mostra la funzione responsabile dell'invio di un workflow JSON al backend di ComfyUI. A ogni richiesta viene associato un identificativo univoco (*client_id*), che consente di tracciare l'esecuzione del job. La funzione restituisce il *prompt_id*, utilizzato successivamente per monitorare lo stato della generazione.

```
def queue_prompt(workflow: dict) -> str:
    payload = {
        "prompt": workflow,
        "client_id": str(uuid.uuid4())
    }

    r = requests.post(
        f"{COMFY}/prompt",
        json=payload,
        timeout=120
    )

    if r.status_code != 200:
        print("COMFYUI ERROR")
        print(r.text)
        r.raise_for_status()

    return r.json()["prompt_id"]
```

Codice 5.3: Invio di un workflow JSON al backend di ComfyUI.

Wait Done

Il codice 5.4 illustra la funzione incaricata di monitorare lo stato di esecuzione di un workflow all'interno di ComfyUI. La funzione implementa un meccanismo di polling che interroga periodicamente il backend utilizzando il `prompt_id` restituito in fase di invio. L'esecuzione viene considerata conclusa nel momento in cui il backend rende disponibili gli output associati al job, garantendo così una sincronizzazione corretta tra la fase di generazione e il recupero dei risultati.

```
def wait_done(prompt_id):
    while True:
        # Interroga lo stato del workflow
        response = http_get("/history/" + prompt_id)

        # Verifica la validità della risposta
        check_response_status(response)

        history = parse_json(response)

        # Controlla se gli output sono disponibili
        if prompt_id in history and history[prompt_id].
   has_outputs():
            return history[prompt_id]

        # Attende prima della prossima interrogazione
        sleep(fixed_interval)
```

Codice 5.4: Pseudocodice per il monitoraggio del completamento di un workflow ComfyUI.

5.2.3 LanPaint in ambiente Python

Oltre i nodi standard per il caricamento delle immagini e del modello su *ComfyUI*, le rispettive pipeline si differenziano in alcuni nodi che caratterizzano l'intero flusso. In questa sezione verranno descritti tali nodi e il rispettivo codice Python. Saranno eslcusi dall'analisi le funzioni ausiliarie che permettono il caricamento dei file necessari dalla memoria, le funzioni di *utilities* e di parsing delle stringhe in quanto non centrali nello studio presente.

I nodi principali della pipeline di *LanPaint* sono principalmente due: *LanPaint MaskBlend* e *LanPaint KSampler*.

Il nodo *LanPaint MaskBlend* 5.5 svolge una funzione cruciale di post-processing all'interno della pipeline LanPaint ed è responsabile della fusione controllata tra l'immagine originale e il risultato generato dal modello di diffusione nella regione definita dalla maschera. Il suo compito è quello di fondere insieme l'immagine generata, l'immagine originale e la maschera al fine di ottenere un risultato coerente e realistico. Il suo ruolo è evitare che l'area modificata appaia come un elemento "incollato" o artificiale rispetto al contesto originale della foto. Infatti senza questo nodo i bordi della maschera potrebbero risultate visibili o introdurre discontinuità rendendo il soggetto generato slegato dalla scena.

```
"78": {
  "inputs": {
    "blend_overlap": 15,
    "image1": [
      "20",
      0
```

```
    ],
    "image2": [
      "8",
      0
    ],
    "mask": [
      "20",
      1
    ]
  },
  "class_type": "LanPaint_MaskBlend",
  "_meta": {
    "title": "LanPaint Mask Blend"
  }
}
```

Codice 5.5: Nodo *LanPaint_MaskBlend* per la fusione finale dell'immagine

Il nodo *LanPaint KSampler* rappresenta il fulcro della pipeline LanPaint. Esso può essere interpretato come un'estensione del classico KSampler utilizzato nei modelli di diffusione, arricchita per supportare operazioni di inpainting e data augmentation contestuale.

A differenza di un KSampler standard, che opera esclusivamente a partire da un rumore iniziale e da un prompt testuale, *LanPaint KSampler* integra simultaneamente tre tipologie di informazione: il prompt testuale, che fornisce indicazioni semantiche sulla modifica da effettuare, l'immagine di input, che funge da riferimento strutturale e visivo e la maschera, che delimita la regione dell'immagine soggetta al processo di rigenerazione.

Questa integrazione consente di esercitare un controllo localizzato sul processo generativo: il rumore viene applicato esclusivamente alle regioni mascherate, mentre le parti non selezionate dell'immagine vengono preservate. In tal modo, *LanPaint KSampler* permette di ottenere variazioni realistiche e coerenti con la scena originale, evitando alterazioni indesiderate del contesto globale.

Inoltre, la presenza di parametri specifici (come il numero di passi LanPaint, il valore di denoise e la modalità di utilizzo del prompt) consente di modulare l'intensità della modifica, rendendo il nodo particolarmente adatto a scenari di data augmentation controllata. La struttura del nodo *LanPaint_Ksampler* è riportata nel codice JSON 5.6.

```
"74": {
  "inputs": {
    "seed": 0,
    "steps": 25,
    "cfg": 5,
    "sampler_name": "dpmpp_2m",
    "scheduler": "karras",
    "denoise": 0.75,
    "LanPaint_NumSteps": 5,
```

```
    "LanPaint_PromptMode": "Image First",
    "Inpainting_mode": "Image Inpainting",
    "model": ["29", 0],
    "positive": ["6", 0],
    "negative": ["7", 0],
    "latent_image": ["66", 0]
  },
  "class_type": "LanPaint_KSampler",
  "_meta": {
    "title": "LanPaint KSampler"
  }
}
```

Codice 5.6: Nodo `LanPaint_KSampler` del workflow LanPaint

Di seguito viene rappresentato il codice della funzione *main* 5.7. È possibile durante l'esecuzione scegliere i parametri di generazione al fine di rendere modulabile e modificabile il risultato.

```
def main():
    # Caricamento del workflow JSON
    workflow = load_json(WORKFLOW_PATH)

    # Inserimento del prompt testuale
    prompt_text = input("Prompt di generazione")

    # Selezione interattiva di immagine e maschera
    image_path = select_file(IMAGES_DIR)
    mask_path  = select_file(MASK_DIR)

    if image_path is None or mask_path is None:
        terminate_execution("File mancanti")

    # Upload delle immagini al backend ComfyUI
    image_name = upload_image(image_path)
    mask_name  = upload_image(mask_path)

    # Accesso ai parametri del nodo LanPaint
    lanpaint_params = workflow["LanPaint_KSampler"].inputs

    # Impostazione interattiva dei parametri principali
    lanpaint_params.seed                = user_input_or_default
(random_seed)
    lanpaint_params.steps               = user_input_or_default
(steps)
    lanpaint_params.cfg                 = user_input_or_default
(cfg)
```

```
    lanpaint_params.sampler_name       = user_input_or_default
(sampler)
    lanpaint_params.scheduler          = user_input_or_default
(scheduler)
    lanpaint_params.denoise             =
user_input_or_default(denoise)
    lanpaint_params.LanPaint_NumSteps  = user_input_or_default
(num_steps)
    lanpaint_params.PromptMode          =
user_input_or_default(prompt_mode)
    lanpaint_params.Inpainting_mode     =
user_input_or_default(inpainting_mode)

    # Impostazione del batch di generazione
    workflow["RepeatLatentBatch"].inputs.amount = (
        user_input_or_default(batch_size)
    )

    # Associazione delle immagini ai nodi LoadImage
    workflow["LoadImage_Image"].inputs.image = image_name
    workflow["LoadImage_Mask"].inputs.image  = mask_name

    # Inserimento del prompt nel nodo CLIP
    workflow["CLIPTextEncode"].inputs.text = prompt_text

    # Invio del workflow al backend ComfyUI
    result = queue_prompt(workflow)

    # Download delle immagini generate
    for output_image in result.outputs:
        download_image(output_image)
```

Codice 5.7: Pseudocodice della funzione principale per l'esecuzione della pipeline LanPaint

5.2.4 ACE++ in ambiente Python

La pipeline basata su ACE++ è progettata per generare in modo sistematico un dataset sintetico a partire da un insieme limitato di scene reali. Il processo è strutturato come una procedura iterativa a più livelli. In primo luogo, il sistema cicla sull'insieme delle scene disponibili. Ciascuna scena rappresenta un contesto visivo distinto (ad esempio una banchina o l'interno di un vagone) ed è associata a un numero variabile di maschere.

Per ogni scena, la pipeline analizza sequenzialmente tutte le maschere disponibili. Ogni maschera definisce una regione specifica dell'immagine all'interno della quale viene inserito un singolo oggetto alla volta, selezionato da una collezione di immagini di riferimento.

Gli oggetti utilizzati sono prelevati da una cartella dedicata e rappresentano le diverse categorie di interesse (ad esempio bagagli, bottiglie o rifiuti). Per ogni tripla scena–maschera–oggetto, il framework *ACE++* viene invocato per generare più istanze dell'immagine finale, variando il seed di generazione. Questo consente di ottenere risultati visivamente diversi ma semanticamente coerenti, aumentando la diversità del dataset sintetico prodotto.

Una difficoltà riscontrata durante lo sviluppo ha riguardato la gestione e la formulazione del prompt. Infatti, dopo alcuni tentativi ed esperimenti, è emerso come l'utilizzo di un prompt generico, valido indistintamente per ogni oggetto da inserire, risultasse poco efficace. Per aggiungere un elemento della scena inizialmente era stato pensato di scrivere, direttamente nel codice Python in modo *hardcoded*, un prompt generico come:

Add this object to the scene.

Tale approccio è risultato inefficiente nell'inserimento degli oggetti, infatti, come riportato nella documentazione di ACE++, il modello si aspetta il nome della categoria dell'oggetto per capirne meglio il contesto. Per superare questa problematica, è stato nominato ogni oggetto all'interno della cartella con il nome appropriato della categoria di appartenenza (es. *bottle.png*) in modo che, tramite codice, si riuscisse ad estrapolare tale informazione per inserirla di volta in volta corretta nel prompt corrispondente. Questo sistema ha permesso di raggiungere risultati nettamente migliori nella generazione.

L'insieme delle immagini generate costituisce quindi un dataset ampliato, in cui la variabilità è introdotta sia a livello spaziale (tramite maschere differenti), sia a livello semantico (tramite oggetti diversi), sia a livello stocastico (tramite generazioni multiple per ciascuna configurazione).

Il codice 5.8 mostra il codice python per la gestione di tale flusso

```python
def main():
    args = parse_args()
    start_total = time.perf_counter()

    os.makedirs(args.out_dir, exist_ok=True)

    print("Loading workflow:", args.workflow)
    with open(args.workflow, "r", encoding="utf-8") as f:
        base_workflow = json.load(f)

    print("Loading prompt template:", args.prompt_file)
    with open(args.prompt_file, "r", encoding="utf-8") as f:
        prompt_template = f.read().strip()

    print("Prompt template loaded:")
    print(prompt_template)
    print("----")

    objects = list_images(args.objects_dir)
```

```
    if not objects:
        raise RuntimeError(f"No objects found in: {args.objects_dir}")

    print(f"Found {len(objects)} objects in {args.objects_dir}")
    for o in objects:
        print("  -", o)

    scene_folders = list_scene_folders(args.scenes_dir)
    if not scene_folders:
        raise RuntimeError(f"No scene folders found in: {args.scenes_dir}")

    print(f"Found {len(scene_folders)} scenes in {args.scenes_dir}")
    for s in scene_folders:
        print("  -", s)

    total_gen = 0

    for scene_name in scene_folders:
        scene_folder = os.path.join(args.scenes_dir, scene_name)
        scene_image_path = find_scene_image(scene_folder)
        masks_dir = find_masks_dir(scene_folder)
        masks = list_images(masks_dir)

        if not masks:
            print(f"Scene '{scene_name}': no masks found, skipping.")
            continue

        scene_ref = upload_image(scene_image_path)
        scene_input = comfy_input(scene_ref)

        for mask_file in masks:
            mask_path = os.path.join(masks_dir, mask_file)
            mask_ref = upload_image(mask_path)
            mask_input = comfy_input(mask_ref)

            for obj_file in objects:
                obj_path = os.path.join(args.objects_dir, obj_file)
                obj_name = object_name_from_file(obj_file)

                obj_ref = upload_image(obj_path)
                obj_input = comfy_input(obj_ref)
```

```
61
62                 out_dir = os.path.join(
63                     args.out_dir,
64                     scene_name,
65                     os.path.splitext(mask_file)[0],
66                     obj_name,
67                 )
68                 os.makedirs(out_dir, exist_ok=True)
69
70                 for i in range(args.n):
71                     seed = random.randint(0, 2**63 - 1)
72                     workflow = deep_copy(base_workflow)
73
74                     final_prompt = prompt_template.format(
    object=obj_name)
75
76                     workflow["21"]["inputs"]["text"] =
    final_prompt
77                     workflow["27"]["inputs"]["image"] =
    scene_input
78                     workflow["23"]["inputs"]["image"] =
    obj_input
79                     workflow["28"]["inputs"]["image"] =
    mask_input
80                     workflow["5"]["inputs"]["seed"] = seed
81                     workflow["25"]["inputs"]["filename_prefix"
    ] = (
82                         f"ACEPP/{scene_name}/{obj_name}/{os.
    path.splitext(mask_file)[0]}"
83                     )
84
85                     pid = queue_prompt(workflow)
86                     result = wait_done(pid)
87
88                     outputs = result.get("outputs", {})
89                     if "25" not in outputs:
90                         raise RuntimeError("Output node 25 not
     found")
91
92                     img_info = outputs["25"]["images"][0]
93                     out_path = os.path.join(
94                         out_dir, f"gen_{i+1:02d}_seed_{seed}.
    png"
95                     )
96                     download_image(img_info, out_path)
97
98                     total_gen += 1
99
```

```
    total_time = time.perf_counter() - start_total
    print("Batch completed")
    print("Total generations:", total_gen)
    print(f"Total time: {total_time:.2f} seconds")
```

Codice 5.8: Funzione principale per l'orchestrazione delle generazioni tramite ComfyUI

5.3 Filtraggio automatico delle generazioni non informative

Durante l'analisi dei risultati prodotti dalle pipeline di inpainting, è emerso un comportamento ricorrente in presenza di maschere di dimensioni molto ridotte. In tali casi, il modello generativo tende frequentemente a non inserire alcun oggetto all'interno della regione mascherata, producendo un'immagine finale identica all'originale.

Questo fenomeno è attribuibile al fatto che, quando l'area disponibile per la generazione risulta troppo limitata, il modello privilegia la preservazione del contesto originale rispetto all'introduzione di nuovi contenuti, interpretando la maschera come non semanticamente rilevante. Di conseguenza, una parte non trascurabile delle immagini generate risulta priva di contributo informativo ai fini dell'ampliamento del dataset.

Dato che l'obiettivo del lavoro è la produzione di campioni sintetici che introducano variazioni effettive e semanticamente significative all'interno delle scene, si è reso necessario definire un criterio di filtraggio automatico per individuare ed eliminare tali generazioni non informative. Sono state quindi scartate le immagini in cui l'inpainting non ha prodotto modifiche apprezzabili rispetto all'immagine originale.

È importante sottolineare che questo filtraggio non rappresenta una soluzione definitiva al problema delle maschere di piccole dimensioni, ma una misura necessaria per garantire la qualità del dataset nella fase attuale del lavoro. Una strategia più strutturata per affrontare tale criticità verrà introdotta in seguito, sfruttando le informazioni di profondità (*depth*) disponibili nel dataset, al fine di imporre vincoli geometrici e dimensionali più coerenti durante la generazione.

5.4 Usare gli infrarossi e la profondità per creare vincoli fisici alla generazione

Come discusso nel Capitolo 4, dedicato alle tecnologie utilizzate, uno degli elementi distintivi del presente lavoro è l'impiego di immagini di profondità acquisite insieme alle immagini RGB della scena.

La disponibilità di informazioni di profondità consente di arricchire il processo di generazione sintetica del dataset introducendo vincoli fisici espliciti, fondamentali per garantire risultati realistici e geometricamente coerenti. Il principale punto di

forza dell'approccio proposto risiede nella possibilità di costruire tali vincoli in modo dinamico e adattivo, direttamente a partire dalla geometria della scena osservata.

5.4.1 Ricavare le distanze reali dall'immagine RGB

Trattiamo di seguito il codice responsabile di determinare le distanze reali tra due punti all'interno dell'immagine RGB (Codice 6.1). L'utente si occupa di selezionare due punti all'interno della scena e il programma restituisce la distanza reale (con un errore nell'ordine di millimetri) proiettata nella scena RGB.

```
def find_closest_3d_point(
    x_color: float,
    y_color: float,
    list_pixel_points_aligned: list
) -> tuple:
    """Restituisce il punto 3D (X,Y,Z) associato al punto di griglia proiettato piu vicino
    alle coordinate normalizzate (x_color, y_color) selezionate dall'utente in RGB."""

def generate_grid(
    top_left: tuple,
    bottom_right: tuple,
    N_hor: int,
    N_vert: int
) -> list:
    """Genera una griglia regolare di punti (x_norm, y_norm) in coordinate normalizzate [0,1]^2
    all'interno del rettangolo definito da top_left e bottom_right."""

def mouse_callback(
    event: int,
    x: int,
    y: int,
    flags: int,
    param
) -> None:
    """Callback OpenCV: converte un click (pixel) nella finestra combinata (RGB+Depth)
    in coordinate normalizzate e aggiorna la lista dei punti target selezionati."""

class TemporalFilter:
    def __init__(self, alpha: float):
        """Inizializza un filtro temporale esponenziale per stabilizzare la mappa di profondita."""

```

```
32      def process(self, frame: np.ndarray) -> np.ndarray:
33          """Applica smoothing temporale combinando il frame
    corrente con quello precedente
34          tramite media pesata (coefficiente alpha)."""
35
36  def parse_point(s: str) -> tuple:
37      """Converte una stringa del tipo '(x,y,z)' in una tupla di
    float (x,y,z);
38      utile per il calcolo della distanza euclidea tra due punti
    3D."""
39
40  def normalize_fixed_range(
41      depth_data: np.ndarray,
42      min_val: float,
43      max_val: float
44  ) -> np.ndarray:
45      """Normalizza una mappa depth in un intervallo [0,255] per
    sola visualizzazione,
46      applicando clipping tra min_val e max_val."""
```

Codice 5.9: Funzioni ausiliarie per la proiezione della griglia depth→RGB e stima 3D

```
1
2   def main():
3       # --- 1) Inizializzazione pipeline Orbbec e configurazione
    stream ---
4       pipeline = Pipeline()
5       config = Config()
6
7       depth_profile_list = pipeline.get_stream_profile_list(
    OBSensorType.DEPTH_SENSOR)
8       if depth_profile_list is None:
9           print("No proper depth profile")
10          return
11      depth_profile = depth_profile_list.
    get_default_video_stream_profile()
12      depth_intrinsics = depth_profile.as_video_stream_profile()
    .get_intrinsic()
13      depth_distortion = depth_profile.as_video_stream_profile()
    .get_distortion()
14      config.enable_stream(depth_profile)
15
16      color_profile_list = pipeline.get_stream_profile_list(
    OBSensorType.COLOR_SENSOR)
17      if color_profile_list is None:
18          print("No proper RGB profile")
19          return
```

```
20      color_profile = color_profile_list.
    get_default_video_stream_profile()
21      color_intrinsics = color_profile.as_video_stream_profile()
    .get_intrinsic()
22      color_distortion = color_profile.as_video_stream_profile()
    .get_distortion()
23      config.enable_stream(color_profile)
24
25      # Parametri extrinseci: trasformazione tra sistema Depth e
        sistema RGB
26      extrinsic = depth_profile.get_extrinsic_to(color_profile)
27
28      # (Opz.) sincronizzazione e filtro di allineamento depth->
    RGB
29      pipeline.enable_frame_sync()
30      pipeline.start(config)
31      align_filter = AlignFilter(align_to_stream=OBStreamType.
    COLOR_STREAM)
32
33      # --- 2) Setup interfaccia (visualizzazione + callback)
    ---
34      cv2.namedWindow("QuickStart Viewer", cv2.WINDOW_NORMAL)
35      cv2.resizeWindow("QuickStart Viewer", window_width,
    window_height)
36      cv2.setMouseCallback("QuickStart Viewer", mouse_callback)
37
38      # Griglia definita in percentuale nel dominio depth
39      list_original_points = generate_grid(top_left,
    bottom_right, N_hor, N_vert)
40
41      # --- 3) Ciclo di acquisizione ---
42      while True:
43          frames_set = pipeline.wait_for_frames(timeout)
44          if frames_set is None:
45              continue
46
47          # Allinea la depth sull'immagine RGB (pixel-wise
    correspondence)
48          aligned_frame = align_filter.process(frames_set)
49          if aligned_frame is None:
50              continue
51          aligned_frames = aligned_frame.as_frame_set()
52          if aligned_frames is None:
53              continue
54
55          color_frame = aligned_frames.get_color_frame()
56          depth_frame_aligned = aligned_frames.get_depth_frame()
57          if (color_frame is None) or (depth_frame_aligned is
```

```
None):
58              continue
59
60          # Depth RAW (non allineata) usata come dominio di
    campionamento della griglia
61          depth_frame_raw = frames_set.get_depth_frame()
62          if depth_frame_raw is None:
63              continue
64
65          # --- 4) Conversione frame -> numpy ---
66          color_image = frame_to_bgr_image(color_frame)
67          color_h, color_w = color_image.shape[:2]
68
69          # Depth RAW (dominio della griglia)
70          wr = depth_frame_raw.get_width()
71          hr = depth_frame_raw.get_height()
72          scale = depth_frame_raw.get_depth_scale()
73          depth_raw = np.frombuffer(depth_frame_raw.get_data(),
    dtype=np.uint16).reshape((hr, wr))
74          depth_raw_mm = depth_raw.astype(np.float32) * scale
75
76          # --- 5) Proiezione griglia: depth(2D) -> RGB(2D) +
    depth(2D)->3D ---
77          list_pixel_points_aligned = []
78          for (x_norm, y_norm) in list_original_points:
79              # punto griglia nel dominio depth (pixel)
80              x_pix = int(np.clip(x_norm * wr, 0, wr - 1))
81              y_pix = int(np.clip(y_norm * hr, 0, hr - 1))
82
83              depth_val = float(depth_raw_mm[y_pix, x_pix])
84              if (not np.isfinite(depth_val)) or depth_val <= 0:
85                  continue
86
87              # (a) depth 2D -> RGB 2D: riproiezione sul piano
    immagine RGB
88              p_rgb = ob.transformation2dto2d(
89                  ob.OBPoint2f(float(x_pix), float(y_pix)),
90                  depth_val,
91                  depth_intrinsics, depth_distortion,
92                  color_intrinsics, color_distortion,
93                  extrinsic
94              )
95
96              # normalizzazione coordinate RGB in [0,1] per
    indipendenza dalla risoluzione
97              x_norm_rgb = p_rgb.x / float(color_w)
98              y_norm_rgb = p_rgb.y / float(color_h)
99
```

```
100             # (b) depth 2D + Z -> 3D: ricostruzione del punto
    nello spazio
101             p_3d = ob.transformation2dto3d(
102                 ob.OBPoint2f(float(x_pix), float(y_pix)),
103                 depth_val,
104                 depth_intrinsics,
105                 extrinsic
106             )
107 
108             # memorizza: pixel RGB + coordinate normalizzate +
     punto 3D
109             list_pixel_points_aligned.append((
110                 int(p_rgb.x), int(p_rgb.y), depth_val,
111                 x_norm_rgb, y_norm_rgb,
112                 float(p_3d.x), float(p_3d.y), float(p_3d.z)
113             ))
114 
115         # --- 6) Visualizzazione: RGB a sinistra, depth a
    destra ---
116         if switch_clip_depth:
117             depth_aligned_np = np.frombuffer(
118                 depth_frame_aligned.get_data(), dtype=np.
    uint16
119             ).reshape((depth_frame_aligned.get_height(),
    depth_frame_aligned.get_width()))
120             depth_aligned_mm = depth_aligned_np.astype(np.
    float32) * depth_frame_aligned.get_depth_scale()
121             depth_aligned_mm = np.where(
122                 (depth_aligned_mm > min_depth) & (
    depth_aligned_mm < max_depth),
123                 depth_aligned_mm, 0
124             )
125 
126         depth_vis = normalize_fixed_range(depth_aligned_mm,
    min_depth, max_depth)
127         depth_vis = cv2.applyColorMap(depth_vis, cv2.
    COLORMAP_INFERNO)
128 
129         rgb_res = cv2.resize(color_image, (window_width // 2,
    window_height))
130         dep_res = cv2.resize(depth_vis, (window_width // 2,
    window_height))
131         combined_image = np.hstack((rgb_res, dep_res))
132 
133         # aggiorna immagine globale per click in coordinate
    normalizzate
134         global current_combined_image
135         current_combined_image = combined_image
```

```
136
137         # disegna (opz.) i punti della griglia proiettati (
    solo per preview)
138         if switch_points_depth or switch_points_RGB:
139             for (_, _, _, x_norm_rgb, y_norm_rgb, _, _, _) in
    list_pixel_points_aligned:
140                 x_rgb = int(x_norm_rgb * (window_width // 2))
141                 y_rgb = int(y_norm_rgb * window_height)
142                 if switch_points_RGB:
143                     cv2.circle(combined_image, (x_rgb, y_rgb),
     1, (0, 255, 0), -1)
144
145         # --- 7) Selezione utente: nearest neighbor sulla
    griglia proiettata ---
146         # target_points viene aggiornato da mouse_callback (
    coordinate normalizzate)
147         if len(target_points) >= 2:
148             p1 = find_closest_3d_point(target_points[0][0],
    target_points[0][1], list_pixel_points_aligned)
149             p2 = find_closest_3d_point(target_points[1][0],
    target_points[1][1], list_pixel_points_aligned)
150
151             if (p1 is not None) and (p2 is not None):
152                 dist_mm = math.dist(p1, p2)
153                 print(f"Distanza stimata: {dist_mm:.2f} mm")
154
155         cv2.imshow("QuickStart Viewer", combined_image)
156
157         # uscita
158         k = cv2.waitKey(1) & 0xFF
159         if k in (ord('q'), ESC_KEY):
160             break
161
162     cv2.destroyAllWindows()
163     pipeline.stop()
```

Codice 5.10: Struttura del ciclo principale (estratto con codice reale)

5.4.2 Creazione della maschera di dimensione prestabilita

A questo punto sono disponibili tutti i componenti necessari per consentire la creazione di una maschera all'interno della scena, mantenendo dimensioni reali e proporzioni coerenti. Il passo successivo consiste nell'adattare il codice presentato in precedenza affinché, fornita in input la dimensione desiderata della maschera (ad esempio 300 mm $\times$ 500 mm), questa venga correttamente inserita nella scena, rispettando le proporzioni spaziali.

Un punto cruciale nell'adattamento del codice per il nostro scopo riguarda il comportamento della profondità all'interno della regione mascherata. In una scena reale, infatti, i pixel contenuti in una stessa area dell'immagine RGB non condividono necessariamente la stessa distanza: la superficie può essere inclinata, possono essere presenti discontinuità (spigoli) oppure oggetti parzialmente sovrapposti. Se si utilizzasse la mappa di profondità originale "così com'è", la maschera risulterebbe vincolata a profondità variabili, rendendo instabile la conversione tra dimensioni metriche e dimensioni in pixel e introducendo distorsioni nella scala finale.

Per risolvere questo problema è stata introdotta una *fake depth map*, ossia una mappa di profondità sintetica costruita imponendo una profondità costante Z_0 all'interno della maschera. Operativamente, quando l'utente clicca un punto sull'immagine RGB, viene stimata la profondità Z_0 associata a quel punto (tramite la procedura di corrispondenza RGB–depth descritta nella sezione precedente). Successivamente, si costruisce una mappa D_{fake} tale che:

$$D_{\text{fake}}(u, v) = \begin{cases} Z_0 & \text{se } (u, v) \in \Omega \\ 0 & \text{altrimenti} \end{cases}$$

dove Ω rappresenta la regione della maschera nel piano immagine. In questo modo la maschera si comporta come un "piano invisibile" (o *muro*) posto alla distanza selezionata, garantendo che l'intera regione sia interpretata a profondità costante.

Questo risulta fondamentale perché consente di proiettare correttamente una maschera definita in unità metriche (ad esempio $300\,\text{mm} \times 500\,\text{mm}$) nel dominio dei pixel, senza essere influenzati dalle variazioni locali della profondità reale.

Il codice 5.11 e 5.12 permette quindi di specificare larghezza e altezza della maschera in millimetri e, cliccando un punto nell'immagine RGB (coperto dalla griglia proiettata, identificabile tramite i punti verdi), generare una *preview* della maschera correttamente scalata. Premendo il tasto `s` è possibile salvare: (i) l'immagine RGB pulita, (ii) la versione RGBA con canale alfa applicato per l'inpainting, (iii) la preview (con griglia e overlay), (iv) la *fake depth map* e (v) un file CSV contenente Z_0, le dimensioni metriche e le coordinate della maschera nel piano immagine.

```
1
2  def rect_px_from_mm_bottom_anchor(
3      xi: int,
4      yi: int,
5      Z0_mm: float,
6      fx: float,
7      fy: float,
8      w_mm: float,
9      h_mm: float,
10     img_w: int,
11     img_h: int
12 ) -> tuple:
13     """
14     Calcola il rettangolo in coordinate pixel a partire da
    dimensioni reali
```

```
    espresse in millimetri. Il rettangolo e' ancorato al punto selezionato
    dall'utente, interpretato come centro del lato inferiore.
    """

def build_mask_and_fake_depth(
    rect: tuple,
    img_w: int,
    img_h: int,
    Z0_mm: float
) -> tuple:
    """
    Costruisce una maschera binaria e una mappa di profondita' artificiale
    (fake depth) in cui tutti i pixel all'interno del rettangolo assumono
    il valore di profondita' Z0.
    """

def build_rgba_mask_gimp_style(
    rgb_bgr: np.ndarray,
    rect_px: tuple
) -> tuple:
    """
    Genera un'immagine RGBA in cui il canale alpha viene impostato a zero
    all'interno del rettangolo, creando un effetto di 'foro' compatibile
    con il processo di inpainting.
    """
```

Codice 5.11: Funzioni ausiliarie per la costruzione della maschera e della fake depth

```

def main():
    # Inizializzazione pipeline e stream RGB / Depth
    pipeline = Pipeline()
    config = Config()
    config.enable_stream(depth_profile)
    config.enable_stream(color_profile)
    pipeline.start(config)

    # Recupero parametri di calibrazione
    color_fx, color_fy = color_intr.fx, color_intr.fy
    extrinsic_d2c = depth_profile.get_extrinsic_to(color_profile)

    while True:
```

```
15          # Acquisizione frame RGB e depth allineata
16          frames = pipeline.wait_for_frames()
17          aligned = align.process(frames).as_frame_set()
18
19          color_img = frame_to_bgr_image(aligned.get_color_frame
    ())
20          depth_mm = extract_depth(aligned.get_depth_frame())
21
22          # Visualizzazione griglia depth proiettata su RGB
23          grid_points_rgb = compute_grid_points_projected_on_rgb
    (...)
24          preview_rgb = draw_grid_points_on_rgb(color_img,
    grid_points_rgb)
25
26          # Gestione input utente (freeze, click, resize
    maschera)
27          if user_click_detected:
28              Z0 = get_depth_safe(depth_mm, xi, yi)
29              rect_px = rect_px_from_mm_bottom_anchor(...)
30              mask, fake_depth = build_mask_and_fake_depth(
    rect_px, ...)
31
32          # Salvataggio dei risultati
33          if save_command:
34              rgba, alpha = build_rgba_mask_gimp_style(color_img
    , rect_px)
35              save_outputs(rgba, fake_depth, metadata)
```

Codice 5.12: Struttura principale della funzione *main*

Capitolo 6

Sperimentazione e validazione

In questo capitolo viene affrontata la fase di sperimentazione e validazione dei risultati ottenuti tramite le tecniche di inpainting descritte nei capitoli precedenti. Lo scopo principale è verificare se gli oggetti inseriti all'interno delle scene risultino sufficientemente verosimili da poter essere identificati da modelli di *object detection* pre-addestrati.

La validazione proposta non ha come obiettivo la valutazione delle prestazioni di specifici modelli di riconoscimento automatico, ma l'analisi della qualità del contenuto sintetico generato. L'utilizzo di modelli di *object detection* come strumento di verifica consente di sostituire, almeno in parte, un processo di valutazione completamente umano, che risulterebbe oneroso in termini di tempo e risorse.

Questo approccio permette di ottenere una misura quantitativa e riproducibile della riconoscibilità degli oggetti generati, offrendo un'alternativa scalabile anche nel caso di dataset sintetici di grandi dimensioni, dove un *assessment* manuale sistematico non sarebbe sostenibile.

6.1 Validazione su dataset pubblici del metodo generativo

Il primo passo nella costruzione del processo di validazione è consistito nell'estrazione di immagini di scene e oggetti a partire da dataset pubblici di riferimento. La scelta di utilizzare dataset pubblici, in particolare COCO, non è stata casuale ma motivata da una precisa esigenza metodologica: i modelli di object detection impiegati per la validazione (YOLOv5, YOLOv8 e OWLViT) sono stati addestrati su dataset di natura analoga. L'utilizzo di immagini appartenenti allo stesso dominio consente di ridurre il rischio che eventuali errori di rilevamento siano imputabili a un *domain shift* e non alla qualità della generazione stessa. In tal modo, la valutazione non mira a testare le prestazioni del modello di detection, bensì a verificare la capacità del metodo generativo di produrre immagini coerenti e riconoscibili secondo modelli allo stato dell'arte, assumendone il corretto funzionamento nel dominio di riferimento.

A tale scopo è stato selezionato il dataset COCO[1], utilizzando esclusivamente le porzioni di validation e test, al fine di evitare possibili forme di contaminazione o bias derivanti dall'impiego di dati potenzialmente utilizzati in fase di addestramento dei modelli di object detection impiegati per la valutazione.

Una prima difficoltà riscontrata riguarda l'estrazione dei singoli oggetti dalle immagini. Il dataset COCO, infatti, è composto prevalentemente da scene complesse in cui gli oggetti di interesse sono inseriti all'interno di contesti articolati e non sono disponibili come immagini isolate. Tuttavia, per l'utilizzo della pipeline ACE++, è necessario disporre di immagini di riferimento che rappresentino l'oggetto in modo più dettagliato e circoscritto.

Per risolvere tale problematica, è stata sfruttata la struttura annotativa del dataset COCO. In particolare, il dataset fornisce, tramite file di annotazione dedicati, informazioni dettagliate sugli oggetti presenti in ciascuna immagine, organizzate per categoria e corredate dalle coordinate delle rispettive *bounding box*. A partire da queste informazioni, è stato sviluppato uno script in Python (Codice 6.1) che, in una prima fase, seleziona le categorie di oggetti di interesse e, successivamente, esegue un'operazione di cropping delle immagini originali guidata dalle bounding box annotate, ottenendo così immagini focalizzate sul singolo oggetto.

```python
from pycocotools.coco import COCO
import os
import cv2
import sys

ANN_PATH = "annotations/instances_val2017.json"
IMG_DIR = "val2017"
OUT_ROOT = "output_crops"

if len(sys.argv) < 2:
    print("Uso: python crop_coco.py <classe1> <classe2> ...")
    sys.exit(1)

categories = sys.argv[1:]
print("Classi richieste:", categories)

coco = COCO(ANN_PATH)

for cat in categories:
    cat_ids = coco.getCatIds(catNms=[cat])
    if not cat_ids:
        print(f"Categoria '{cat}' non trovata")
        continue

    ann_ids = coco.getAnnIds(catIds=cat_ids)
    anns = coco.loadAnns(ann_ids)
```

[1] https://cocodataset.org

```
27
28      out_dir = os.path.join(OUT_ROOT, cat)
29      os.makedirs(out_dir, exist_ok=True)
30
31      print(f"{cat}: {len(anns)} oggetti")
32
33      for i, ann in enumerate(anns):
34          img_info = coco.loadImgs(ann["image_id"])[0]
35          img_path = os.path.join(IMG_DIR, img_info["file_name"
    ])
36
37          img = cv2.imread(img_path)
38          if img is None:
39              continue
40
41          x, y, w, h = map(int, ann["bbox"])
42          crop = img[y:y+h, x:x+w]
43
44          out_name = f"{img_info['id']}_{i}.png"
45          cv2.imwrite(os.path.join(out_dir, out_name), crop)
46
47      print(f"Salvati crop per {cat}\n")
```

Codice 6.1: Script Python per l'estrazione e il crop di oggetti dal dataset COCO

6.1.1 Validazione tramite IA

Una volta completata l'estrazione degli oggetti a partire dal dataset pubblico selezionato, il passo successivo ha riguardato la definizione di una procedura di validazione, finalizzata a stimare in che misura gli oggetti inseriti tramite inpainting risultassero effettivamente riconoscibili da modelli di object detection pre-addestrati.

A tale scopo, è stata condotta un'analisi statistica basata sul numero di istanze correttamente individuate dai modelli di riconoscimento automatico, considerando come criterio di successo la capacità del modello di identificare la classe dell'oggetto inserito all'interno della scena.

La selezione dei modelli di object detection è stata effettuata tenendo conto di due fattori principali: da un lato, le prestazioni riportate in letteratura e l'ampia diffusione nella comunità di computer vision; dall'altro, i vincoli computazionali imposti dall'ambiente di sperimentazione, che hanno reso necessario l'utilizzo di modelli in grado di eseguire inferenza in modo efficiente sulla workstation messa a disposizione durante il tirocinio.

I modelli scelti quindi sono stati:

- YOLOv8 (You Only Look Once): È stato scelto in quanto rappresenta lo stato dell'arte tra i modelli YOLO in termini di accuratezza, stabilità e velocità di inferenza. Il modello risulta addestrato sul dataset pubblico COCO.

- YOLOv5 (You Only Look Once): Incluso come baseline storica, è l'architettura legacy di YOLO. Ciò ha permesso un confronto tra un detector moderno e uno consolidato ma meno recente, permettendo di valutare se il riconoscimento degli oggetti dipende dalla capacita del modello oppure se il realismo delle immagini è sufficiente anche per modelli più datati. Anche questo modello risulta addestrato su COCO.

- OWL-VIT (Open-World Localization Vision Transformer): È un modello di object detection open-vocabulary basato su architettura Transformer, in grado di individuare oggetti a partire da descrizioni testuali, senza essere limitato a un insieme chiuso di classi come nel caso dei modelli YOLO.

Gli oggetti considerati per l'analisi statistica appartengono alle categorie *Backpack*, *Book*, *Bottle*, *Cup*, *Handbag* e *Suitcase*. Sono state poi prese tredici scene dal dataset e per ogni scena sono state prodotte manualmente, tramite un software di modifica di immagini, più maschere in punti diversi. Ogni oggetto in ogni scena è stato generato dieci volte facendo variare il seed in maniera casuale. La produzione finale conta 14.100 immagini.

Dopo la scelta dei modelli, questi sono stati integrati in uno script Python che ne permettesse di fare inferenza.

Viene riportato di seguito il codice utilizzato per la generazione del file CSV su cui è stato poi sviluppato un grafico per valutare i risultati.

Nel codice riportato in 6.2 è illustrata la procedura necessaria per l'utilizzo del modello OWL-ViT. Analogamente, in 6.3 è mostrato l'esempio di utilizzo del modello YOLO, che risulta di più immediata applicazione grazie alla libreria dedicata `ultralytics`.

Per garantire sia l'analisi quantitativa globale sia la tracciabilità delle singole valutazioni, la pipeline di validazione produce due file distinti: un file di riepilogo aggregato per classe e un file dettagliato contenente il risultato di ogni singola immagine (Vedi codice 6.4).

```
import torch
from PIL import Image
from transformers import (
    OwlViTProcessor,
    OwlViTForObjectDetection
)

class OWLViTDetector:
    def __init__(self, conf=0.25):
        self.device = (
            "cuda"
            if torch.cuda.is_available()
            else "cpu"
        )

```

```
        self.processor = OwlViTProcessor.from_pretrained(
            "google/owlvit-base-patch32"
        )
        self.model = OwlViTForObjectDetection.from_pretrained(
            "google/owlvit-base-patch32"
        ).to(self.device)

        self.conf = conf

    def detect(self, img_path, expected_object):
        image = Image.open(img_path).convert("RGB")

        texts = [[expected_object]]

        inputs = self.processor(
            text=texts,
            images=image,
            return_tensors="pt"
        ).to(self.device)

        with torch.no_grad():
            outputs = self.model(**inputs)

        target_sizes = torch.tensor(
            [image.size[::-1]]
        ).to(self.device)

        results = self.processor.post_process_object_detection
    (
            outputs,
            threshold=self.conf,
            target_sizes=target_sizes
        )[0]

        labels = [
            expected_object.lower()
            for _ in results["scores"]
        ]
        confs = results["scores"].tolist()

        return labels, confs
```

Codice 6.2: Implementazione del detector OWL-ViT per il riconoscimento zero-shot degli oggetti

```
from ultralytics import YOLO

```

```python
class YOLODetector:
    def __init__(self, model_path, conf=0.25):
        self.model = YOLO(model_path)
        self.conf = conf

    def detect(self, img_path, expected_object=None):
        results = self.model(
            img_path,
            conf=self.conf,
            verbose=False
        )[0]

        if results.boxes is None:
            return [], []

        names = results.names

        labels = [
            names[int(i)].lower()
            for i in results.boxes.cls.tolist()
        ]
        confs = results.boxes.conf.tolist()

        return labels, confs
```

Codice 6.3: Implementazione del detector YOLO per il riconoscimento supervisionato degli oggetti

```python
import os
import csv
from collections import defaultdict

from detectors import YOLODetector, OWLViTDetector


RESULTS_DIR = "risultati"
OUT_ROOT = "eval_csv"
CONF_THRESHOLD = 0.25

DETECTORS = {
    "yolov8": YOLODetector("yolov8n.pt", CONF_THRESHOLD),
    "yolov5": YOLODetector("yolov5s.pt", CONF_THRESHOLD),
    "owlvit": OWLViTDetector(CONF_THRESHOLD),
}

for detector_name, detector in DETECTORS.items():
    print(f"\nValutazione modello: {detector_name}")
```

```
20
21      stats = defaultdict(lambda: {
22          "total": 0,
23          "found": 0,
24          "not_found": 0
25      })
26      detail_rows = []
27
28      for scene_name in os.listdir(RESULTS_DIR):
29          scene_dir = os.path.join(RESULTS_DIR, scene_name)
30          if not os.path.isdir(scene_dir):
31              continue
32
33          for mask_name in os.listdir(scene_dir):
34              mask_dir = os.path.join(scene_dir, mask_name)
35              if not os.path.isdir(mask_dir):
36                  continue
37
38              for object_name in os.listdir(mask_dir):
39                  object_dir = os.path.join(mask_dir,
    object_name)
40                  if not os.path.isdir(object_dir):
41                      continue
42
43                  expected_object = object_name.lower()
44
45                  for filename in os.listdir(object_dir):
46                      if not filename.lower().endswith(
47                          (".png", ".jpg", ".jpeg", ".webp")
48                      ):
49                          continue
50
51                      img_path = os.path.join(object_dir,
    filename)
52                      stats[expected_object]["total"] += 1
53
54                      detected, confs = detector.detect(
55                          img_path, expected_object
56                      )
57
58                      if expected_object in detected:
59                          stats[expected_object]["found"] += 1
60                          detected_flag = "yes"
61                          conf = max(
62                              c for c, d in zip(confs, detected)
63                              if d == expected_object
64                          )
65                      else:
```

```
                        stats[expected_object]["not_found"] +=
    1
                        detected_flag = "no"
                        conf = 0.0

                    detail_rows.append([
                        scene_name,
                        mask_name,
                        expected_object,
                        filename,
                        detected_flag,
                        round(conf, 3)
                    ])



    out_dir = os.path.join(OUT_ROOT, detector_name)
    os.makedirs(out_dir, exist_ok=True)

    with open(
        os.path.join(out_dir, "summary.csv"),
        "w",
        newline="",
        encoding="utf-8"
    ) as f:
        w = csv.writer(f)
        w.writerow([
            "object",
            "total",
            "found",
            "not_found",
            "success_rate"
        ])
        for obj, s in stats.items():
            rate = (
                s["found"] / s["total"] * 100
                if s["total"] else 0
            )
            w.writerow([
                obj,
                s["total"],
                s["found"],
                s["not_found"],
                round(rate, 2)
            ])

    with open(
        os.path.join(out_dir, "details.csv"),
```

```
        "w",
        newline="",
        encoding="utf-8"
    ) as f:
        w = csv.writer(f)
        w.writerow([
            "scene",
            "mask",
            "object",
            "image",
            "detected",
            "confidence"
        ])
        w.writerows(detail_rows)

    print(f"CSV salvati in {out_dir}")
```

Codice 6.4: Pipeline di validazione automatica e generazione delle statistiche tramite modelli di object detection

6.1.2 Analisi statistica preliminare delle generazioni

La Figura 6.1 riporta il *success rate* dei modelli di object detection considerati, definito come la percentuale di immagini in cui il modello è riuscito a identificare correttamente un oggetto appartenente alla classe attesa, rispetto al numero totale di immagini generate contenenti un oggetto di quella classe. L'altezza delle barre rappresenta dunque il tasso di rilevamento percentuale per ciascuna coppia modello–oggetto, mentre il valore numerico riportato sopra ogni barra indica il numero assoluto di rilevamenti corretti ottenuti.

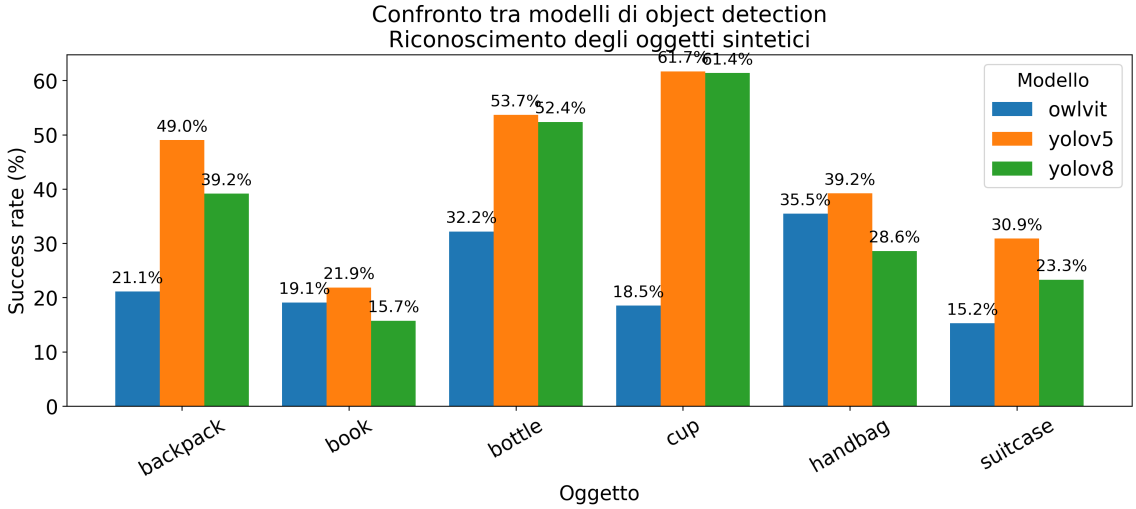


Figura 6.1: Confronto tra i modelli di object detection pre-addestrati in termini di tasso di riconoscimento degli oggetti inseriti tramite inpainting.

Un'ulteriore informazione restituita dai modelli di object detection utilizzati, e che risulta particolarmente rilevante ai fini dell'analisi, è il *detection confidence score*, cioè un valore reale compreso tra 0 e 1 che esprime il grado di confidenza del modello nell'associare una determinata predizione all'oggetto ricercato, e che viene utilizzato per filtrare le rilevazioni meno affidabili tramite una soglia minima prefissata

La Figura 6.2 mostra la confidence media restituita dai modelli nei soli casi di rilevamento. Tale misura non rappresenta un'accuratezza del modello, ma il grado di certezza con cui l'oggetto aggiunto viene riconosciuto. I risultati evidenziano come YOLOv5 e YOLOv8 tendano a fornire confidence più elevate rispetto a OWL-ViT, in particolare per categorie come bottle e cup, caratterizzate da una struttura visiva ben definita. Al contrario, oggetti più variabili come book o handbag mostrano valori medi inferiori e una maggiore dispersione, indicando una maggiore ambiguità visiva nelle immagini generate.

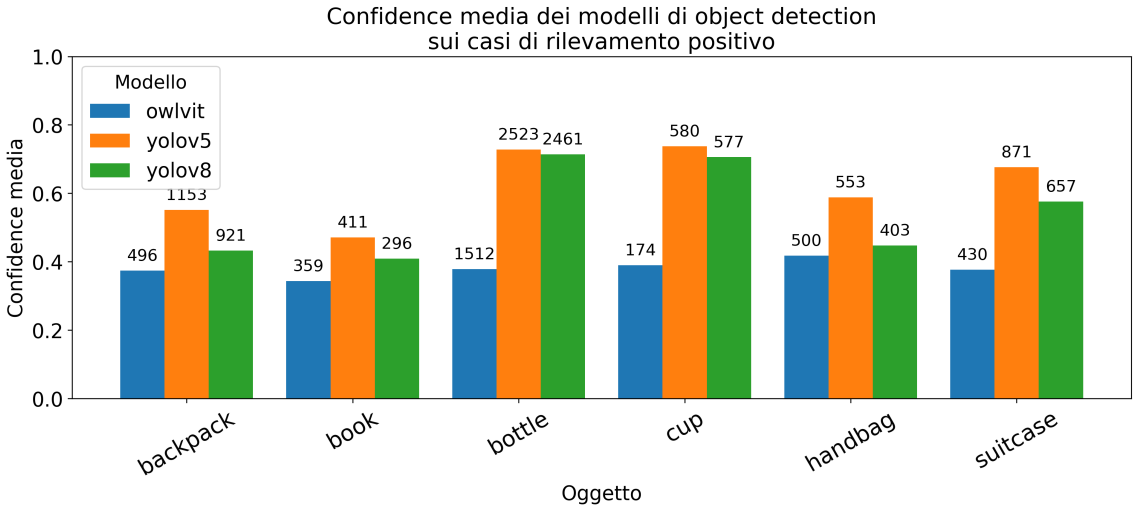


Figura 6.2: Confronto tra la *confidence* risultante per le immagini dei tre modelli scelti

I risultati ottenuti mostrano che i modelli di *detection* faticano a riconoscere gli oggetti inseriti nelle immagini generate. In alcuni casi, i valori della *confidence* risultano bassi o intermedi, indicando un'incertezza del modello o l'assenza di una predizione affidabile. Questo comportamento si può attribuire a diversi fattori, tra cui la qualità visiva delle immagini generate, la coerenza geometrica dell'oggetto rispetto alla scena, o la presenza di artefatti introdotti dalla generazione.

È importante dire anche che l'assenza di riconoscimento sistematico e ad alta *confidence* suggerisce che le immagini sintetiche non vengano interpretate dai modelli come esempi "banali" o facilmente assimilabili a istanze già note.

Nel caso in esame, le differenze osservate tra i modelli possono essere attribuite, almeno in parte, alle differenti categorie e rappresentazioni semantiche apprese durante la fase di training. Modelli come OWL-ViT, basati su rappresentazioni visivo–testuali, non sono necessariamente addestrati sulle stesse classi esplicite utilizzate dai modelli YOLO, i quali fanno riferimento a un insieme di categorie predefinite.

Di conseguenza, l'assenza di riconoscimento per alcune classi può riflettere un disallineamento semantico tra le categorie generate e quelle effettivamente apprese dal modello, piuttosto che un limite intrinseco delle immagini sintetiche.

6.1.3 Analisi statistica delle immagini filtrate

Come precedentemente riportato, un passo implementativo che può migliorare i risultati consiste nella rimozione delle immagini che rappresentano esclusivamente rumore, oppure di immagini in cui, all'interno delle maschere, non è stato generato alcun contenuto significativo.

Ripulire il dataset da queste situazioni non è stato banale. Il primo approccio considerato si basa sul confronto diretto tra l'immagine originale della scena e l'immagine generata, assumendo che una generazione valida debba introdurre una modifica visiva apprezzabile. A tale scopo sono stati sperimentati sia confronti *pixel-wise* basati sulla differenza assoluta tra immagini, sia l'utilizzo di metriche di similarità strutturale (SSIM). L'idea alla base di questi metodi è che una generazione che non modifica in modo significativo la scena originale possa essere considerata poco informativa e quindi scartata.

L'applicazione di questi metodi però non ha portato al risultato voluto, presentando diverse limitazioni. Infatti venivano scartati anche risultati di oggetti molto piccoli anche se corretti, inoltre questi metodi sono particolarmente suscettibili ai cambi di luminosità, generando falsi positivi o falsi negativi. Di conseguenza, immagini in cui l'oggetto era correttamente inserito ma occupava una porzione limitata della scena venivano talvolta erroneamente scartate. Per superare questi limiti, il confronto è stato successivamente limitato alle regioni di interesse, definite tramite le maschere. Quindi si andava a valutare non più l'intera immagine ma solamente la zona modificata. Nonostante un primo miglioramento introdotto da questo approccio, ancora vinivano scartate immagini corrette e non scartate immagini che presentavano artefatti. Questa esigenza ha motivato l'adozione di modelli multimodali come CLIP, in grado di confrontare immagini in uno spazio di rappresentazione semantica condiviso, superando i limiti dei metodi basati esclusivamente su similarità visiva. Nel contesto di questo lavoro, CLIP è stato utilizzato per confrontare direttamente l'immagine originale della scena e l'immagine generata, limitando l'analisi alla regione definita dalla maschera di inpainting. In particolare, per ciascuna generazione, viene estratta una porzione dell'immagine corrispondente all'area mascherata (con l'aggiunta di un margine di contesto), sia dalla scena originale sia dall'immagine sintetica. Le due regioni vengono quindi proiettate nello spazio di embedding di CLIP e confrontate tramite similarità coseno. L'ipotesi alla base di questo approccio è che, qualora l'oggetto non venga effettivamente inserito o risulti semanticamente incoerente, la rappresentazione CLIP dell'immagine generata rimanga fortemente simile a quella della scena originale. Al contrario, una modifica semanticamente rilevante dovrebbe produrre una riduzione della similarità tra le due rappresentazioni. Le immagini per cui la similarità CLIP supera una soglia prefissata vengono quindi considerate poco informative e scartate, mentre le generazioni che introducono una variazione semantica significativa vengono mantenute nel dataset finale. Una volta

implementata la procedura, presente nel codice 6.5, lo step successivo è stato tarare la soglia di scarto in modo da permettere l'eliminazione di immagini che non includevano nuove informazioni al dataset ma allo stesso tempo non renderlo troppo aggressivo nella scelta delle immagini da scartare.

```
import os
import shutil
import torch
import numpy as np
from PIL import Image
from tqdm import tqdm
from transformers import CLIPProcessor, CLIPModel

BASE_DIR = os.path.dirname(os.path.abspath(__file__))

RESULTS_ROOT = "risultati"
SCENE_ROOT   = os.path.join(BASE_DIR, "..", "Validazione", "scene")
REJECTED_DIR = "risultati_scartati"

DRY_RUN = False

CLIP_MODEL_NAME = "openai/clip-vit-base-patch32"
CLIP_SIM_THRESHOLD = 0.98

MASK_THRESHOLD = 128
PADDING = 20

IMG_EXT = (".png", ".jpg", ".jpeg", ".webp")

device = "cuda" if torch.cuda.is_available() else "cpu"

model = CLIPModel.from_pretrained(CLIP_MODEL_NAME).to(device)
processor = CLIPProcessor.from_pretrained(CLIP_MODEL_NAME)

model.eval()

def find_scene_image(scene_dir):
    for ext in IMG_EXT:
        p = os.path.join(scene_dir, f"scena{ext}")
        if os.path.isfile(p):
            return p
    raise FileNotFoundError(f"Scena non trovata in {scene_dir}")

def find_mask(scene_name, mask_name):
    p = os.path.join(
        SCENE_ROOT, scene_name, "maschere", f"{mask_name}.png"
```

```
    )
    if not os.path.isfile(p):
        raise FileNotFoundError(f"Maschera non trovata: {p}")
    return p

def load_image(path):
    return Image.open(path).convert("RGB")

def load_mask_bbox(mask_path, size):

    mask = Image.open(mask_path).convert("L").resize(size, Image.NEAREST)
    mask_np = np.array(mask) > MASK_THRESHOLD

    if not np.any(mask_np):
        return None

    ys, xs = np.where(mask_np)
    x1, x2 = xs.min(), xs.max()
    y1, y2 = ys.min(), ys.max()

    x1 = max(0, x1 - PADDING)
    y1 = max(0, y1 - PADDING)
    x2 = min(size[0], x2 + PADDING)
    y2 = min(size[1], y2 + PADDING)

    return (x1, y1, x2, y2)

@torch.no_grad()
def clip_embedding(image):
    inputs = processor(images=image, return_tensors="pt").to(device)
    emb = model.get_image_features(**inputs)
    emb = emb / emb.norm(dim=-1, keepdim=True)
    return emb

def cosine_similarity(a, b):
    return float((a @ b.T).item())

def unique_name(scene, mask, obj, fname):
    return f"{scene}_mask{mask}_{obj}_{fname}"

def main():
    os.makedirs(REJECTED_DIR, exist_ok=True)

    kept = removed = skipped = 0
    tasks = []

```

```
    # Raccolta immagini generate
    for scene_name in os.listdir(RESULTS_ROOT):
        scene_dir = os.path.join(RESULTS_ROOT, scene_name)
        if not os.path.isdir(scene_dir):
            continue

        for mask_name in os.listdir(scene_dir):
            mask_dir = os.path.join(scene_dir, mask_name)
            if not os.path.isdir(mask_dir):
                continue

            for obj_name in os.listdir(mask_dir):
                obj_dir = os.path.join(mask_dir, obj_name)
                if not os.path.isdir(obj_dir):
                    continue

                for f in os.listdir(obj_dir):
                    if f.lower().endswith(IMG_EXT):
                        tasks.append(
                            (scene_name, mask_name, obj_name,
                             os.path.join(obj_dir, f))
                        )

    if not tasks:
        print("Nessuna immagine trovata.")
        return

    pbar = tqdm(tasks, desc="CLIP filtering (masked)", unit="
img")
    scene_cache = {}

    for scene_name, mask_name, obj_name, gen_path in pbar:
        try:
            scene_dir = os.path.join(RESULTS_ROOT, scene_name)

            if scene_name not in scene_cache:
                scene_path = find_scene_image(scene_dir)
                scene_img = load_image(scene_path)
                scene_cache[scene_name] = scene_img

            scene_img = scene_cache[scene_name]
            gen_img   = load_image(gen_path)

            bbox = load_mask_bbox(
                find_mask(scene_name, mask_name),
                scene_img.size
            )

```

```
            if bbox is None:
                skipped += 1
                continue

            scene_crop = scene_img.crop(bbox)
            gen_crop   = gen_img.crop(bbox)

            emb_scene = clip_embedding(scene_crop)
            emb_gen   = clip_embedding(gen_crop)

            sim = cosine_similarity(emb_scene, emb_gen)
            pbar.set_postfix_str(f"sim={sim:.3f}")

            if sim >= CLIP_SIM_THRESHOLD:
                removed += 1
                dst_name = unique_name(
                    scene_name, mask_name, obj_name,
                    os.path.basename(gen_path)
                )
                dst_path = os.path.join(REJECTED_DIR, dst_name
    )

                if DRY_RUN:
                    shutil.copy2(gen_path, dst_path)
                else:
                    os.remove(gen_path)
            else:
                kept += 1

        except Exception:
            skipped += 1

    print("Filtraggio CLIP + maschera completato")

if __name__ == "__main__":
    main()
```

Codice 6.5: Filtraggio semantico delle immagini sintetiche mediante CLIP e maschere di inpainting

Il programma ha rilevato circa 6000 immagini che potevano risultare non corrette, dopodiché è stata riprodotta la stima sul dataset che ha prodotto i risultati in figure 6.3 e 6.4, che sono le analoghe delle figure 6.1 e 6.2 ma limitate alle sole circa 8000 immagini che hanno superato la scrematura.

Come si può evincere dai grafici sopra riportati, i risultati dopo l'applicazione delle fasi di filtraggio mostrano valori comparabili a quelli osservati sul dataset non filtrato, con variazioni contenute. Tale comportamento è coerente con l'obiettivo del filtraggio adottato, che non mira a semplificare il problema di object detection, ma

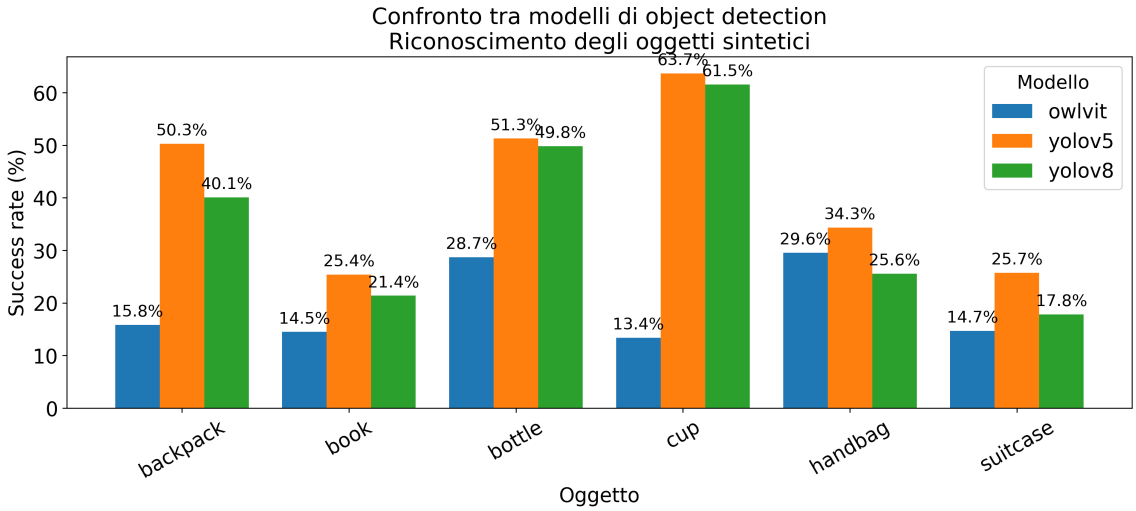


Figura 6.3: Confronto tra il *success rate* per le immagini dei tre modelli scelti dopo il filtraggio

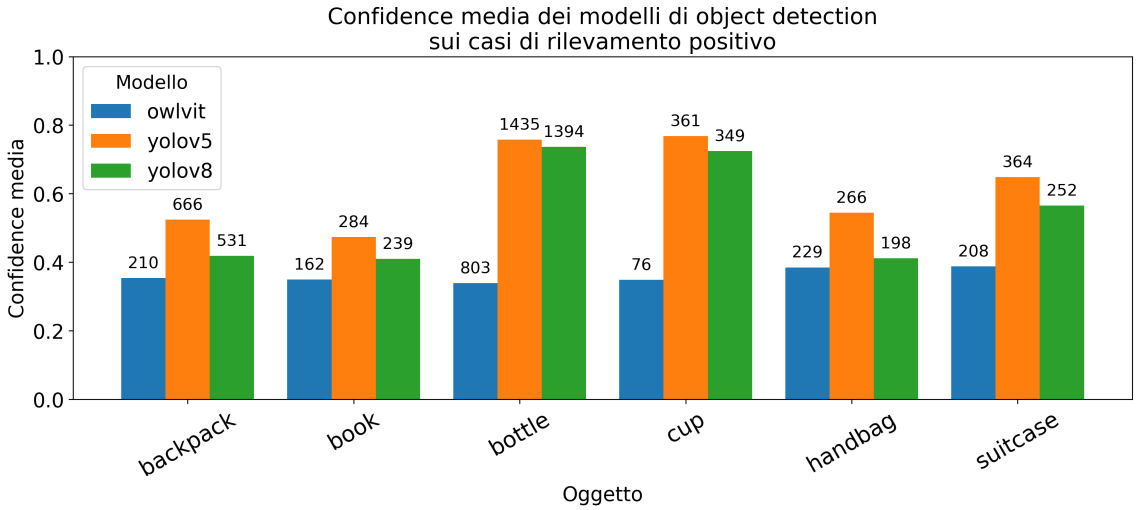


Figura 6.4: Confronto tra la *confidence* risultante per le immagini dei tre modelli scelti dopo il filtraggio

a eliminare generazioni fallite. In particolare, le immagini già caratterizzate da una chiara presenza dell'oggetto risultano riconoscibili anche prima del filtraggio, mentre i casi ambigui o non informativi vengono semplicemente rimossi. Di conseguenza, il filtraggio contribuisce a ridurre il rumore del set di immagini senza alterarne significativamente la distribuzione.

6.1.4 Analisi statistica delle immagini generate con maschera dinamica

Riteniamo inoltre interessante riportare i risultati ottenuti dalla generazione di immagini in cui è stata applicata una maschera dinamica definita rispettando le corrette proporzioni metriche dell'oggetto da inserire. In particolare, vengono analizzati il tasso di successo del riconoscimento e la confidence media associati alle immagini generate utilizzando maschere dimensionalmente compatibili con l'oggetto desiderato, riportati rispettivamente in Figura 6.5 e in Figura 6.6.

Non è stato possibile applicare tale procedura né la relativa validazione alle scene utilizzate nel paragrafo precedente, tratte dal dataset COCO, poiché il processo di definizione delle maschere dinamiche si basa sull'informazione di profondità (*depth*), valore che nel nostro approccio viene acquisito direttamente dalla telecamera. Si è reso pertanto necessario acquisire nuove immagini direttamente mediante la telecamera Orbbec, al fine di disporre delle mappe di profondità richieste per la corretta costruzione delle maschere.

Le dimensioni metriche degli oggetti da inserire vengono inizialmente ottenute a partire da un file di metadati in formato JSON, che contiene le misure nominali associate a ciascuna categoria. Nel dataset originale tali dimensioni non vengono assunte come valori assoluti, ma vengono anch'esse ricavate dall'informazione di profondità, poiché anche per gli oggetti di riferimento è disponibile la corrispondente mappa di *depth*.

Lo scopo del sistema di mascheramento dinamico è di vincolare il processo di generazione a monte, evitando la definizione di maschere incompatibili con le dimensioni dell'oggetto da inserire. In particolare, il sistema impedisce la generazione di immagini in cui lo spazio disponibile risulti eccessivamente ridotto o, al contrario, sproporzionatamente ampio rispetto all'oggetto desiderato, configurazioni che sono note condurre con elevata probabilità a risultati errati o semanticamente incoerenti.

Attraverso questa strategia è possibile ridurre significativamente il numero di generazioni inutili, ottimizzando il costo computazionale complessivo e migliorando l'efficienza del sistema, soprattutto in scenari di utilizzo su larga scala.

6.1.5 Confronto tra generazione con maschera dinamica e scene COCO

Dall'analisi comparativa delle *confusion matrix* tra valutazione umana e riconoscimento automatico, riportate in Figura 6.7, emergono le seguenti considerazioni.

- **Maggiore numero di immagini valide nel metodo con maschera dinamica.** Dalla Figura 6.7(a) si osserva come la somma della prima colonna (ossia il numero totale di immagini giudicate corrette dall'annotazione umana) risulti significativamente superiore rispetto al caso COCO (72.2% contro 58.1%). Questo indica che l'approccio basato su maschere definite tramite informazione di profondità produce un numero maggiore di immagini semanticamente coe-

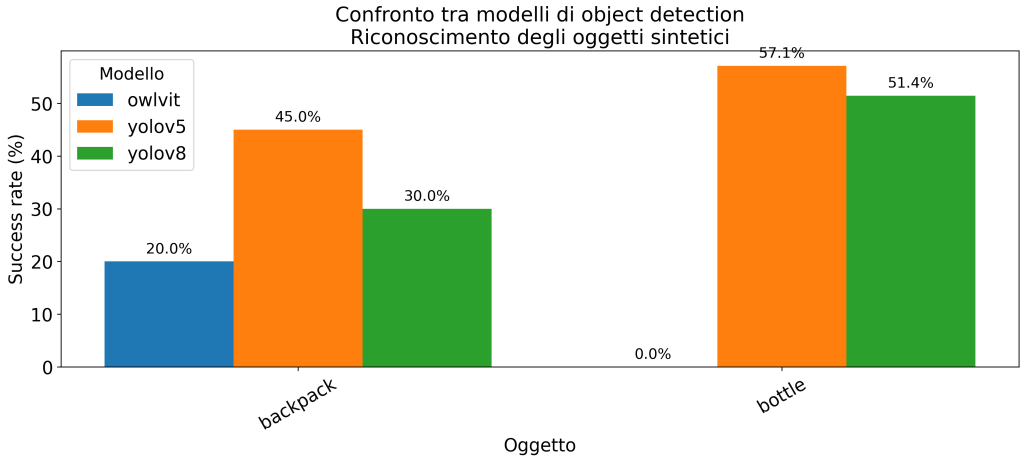


Figura 6.5: Tasso di successo del rilevamento sulle immagini generate mediante maschera dinamica.

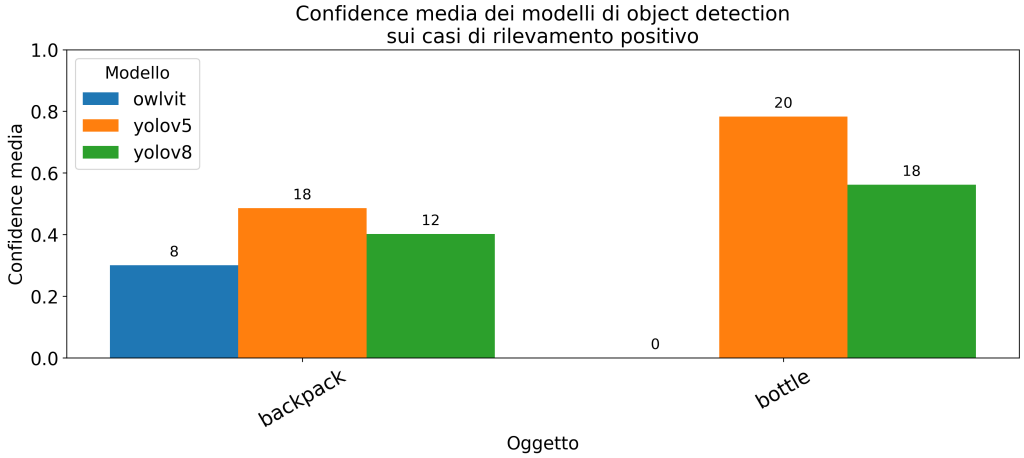
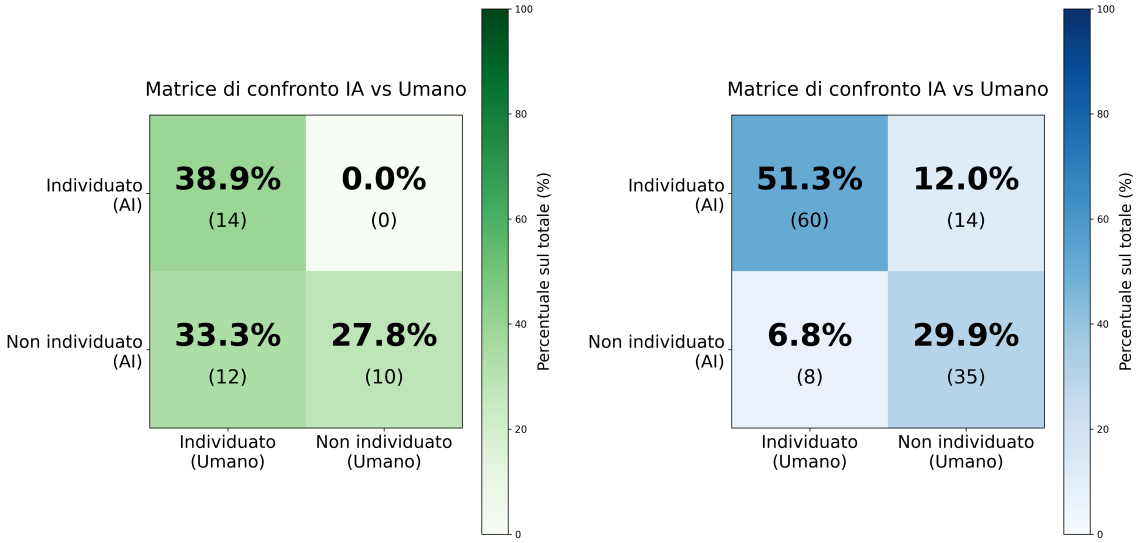


Figura 6.6: Confidence media dei modelli sui casi di rilevamento positivo per immagini generate mediante maschera dinamica.

renti, evidenziando una migliore compatibilità tra oggetto generato e contesto della scena.

- **Maggiore accordo umano–IA nel dominio COCO.** Dalla Figura 6.7(b) si nota che il valore associato alla cella "Umano sì – IA sì" risulta più elevato nel caso delle scene COCO. Tale comportamento è coerente con le aspettative, poiché i modelli di object detection impiegati sono stati addestrati proprio sul dataset COCO. Le immagini appartenenti allo stesso dominio di addestramento risultano quindi più favorevoli al riconoscimento automatico. Questo suggerisce che l'approccio con maschera dinamica, operando su un dominio differente rispetto al dataset di training, possa richiedere un adattamento del modello o una strategia di validazione più specifica.

- **Presenza di falsi positivi nel caso COCO.** Nella Figura 6.7(b), relativa



(a) Generazione con maschera dinamica (b) Scene tratte dal dataset COCO

Figura 6.7: Confronto tra accordo umano–IA in due esperimenti distinti: (a) generazione con maschera dinamica (36 immagini analizzate); (b) scene tratte dal dataset COCO (117 immagini analizzate). Le percentuali riportate in ciascuna cella sono calcolate rispetto al totale delle immagini dell'esperimento considerato. I valori numerici tra parentesi indicano il numero assoluto di campioni corrispondenti a ciascuna combinazione di valutazione umano–IA.

alle scene COCO, la cella "Umano no – IA sì" assume un valore pari al 12%, mentre nella Figura 6.7(a), relativa alla generazione con maschera dinamica, tale valore risulta nullo.

Questo significa che, nel caso COCO, il modello di object detection individua l'oggetto anche in immagini per le quali la valutazione umana relativa all'oggetto generato è negativa.

Una possibile spiegazione è la presenza, nelle immagini originali del dataset COCO, di oggetti appartenenti alla stessa categoria già presenti nella scena. In assenza di un vincolo spaziale, il modello può quindi rilevare un oggetto semanticamente corretto ma distinto da quello sintetico inserito.

Per ridurre tale ambiguità sarebbe opportuno introdurre un criterio spaziale basato sull'Intersection over Union (IoU), verificando che il bounding box rilevato coincida con la regione di inserimento dell'oggetto sintetico. L'adozione di tale misura consentirebbe di distinguere tra rilevamenti corretti ma non pertinenti e rilevamenti realmente associati all'oggetto generato, rappresentando un possibile miglioramento metodologico per sviluppi futuri.

Capitolo 7

Conclusioni

7.1 Conclusioni scientifiche

In questo lavoro è stata progettata e implementata una pipeline completa per la generazione e validazione automatica di immagini sintetiche mediante modelli di diffusione. Il sistema sviluppato consente l'inserimento controllato di oggetti in scene esistenti, attraverso la definizione di maschere dinamiche basate sulle dimensioni di profondità acquisita da sensore RGB-D. Parallelamente, è stato realizzato un modulo di valutazione automatica fondato su modelli di object detection, finalizzato alla verifica quantitativa della riconoscibilità degli oggetti generati. L'intero framework integra fase generativa, fase di controllo e analisi statistica dei risultati, permettendo una valutazione sistematica della qualità del dataset prodotto.

Il sistema è attualmente impiegato per l'assemblaggio di un dataset su larga scala, composto da circa 30 000 immagini sintetiche generate a partire da oltre 300 oggetti acquisiti in configurazione multivista. Il dataset risultante è destinato a supportare attività di addestramento, validazione e test di modelli di riconoscimento visivo.

Il lavoro si inserisce in una progettualità più ampia (*Fostering Artificial Intelligence Trust for Humans N.101135932*) orientata alla costruzione di dataset sintetici per applicazioni di computer vision. In tale contesto, la pipeline sviluppata rappresenta il componente responsabile della la generazione controllata di immagini, contribuendo alla creazione di dati utili per addestramento, validazione e test di modelli di riconoscimento visivo.

Dal punto di vista metodologico, il contributo principale riguarda l'introduzione di un sistema di mascheramento dinamico basato su vincoli metrici derivati dalla profondità della scena (calcolata mediante sensore infrarosso coregistrato). Tale approccio consente di vincolare a priori la fase generativa, evitando la creazione di configurazioni geometricamente incoerenti quali oggetti di misure non realistiche oppure generazioni errate degli oggetti.

Per la validazione automatica delle immagini generate sono stati impiegati tre modelli di object detection: due modelli YOLO pre-addestrati su COCO (YOLOv5 e YOLOv8) e un modello OWLViT, non specificamente addestrato su COCO ma basato su un'architettura vision–language generalista.

I risultati sperimentali evidenziano come la generazione con maschera dinamica produca una quota più elevata di immagini giudicate valide rispetto alla generazione su scene COCO campionate casualmente. Nel campione analizzato, il tasso di generazioni corrette risulta superiore nel caso della generazione dinamica (circa 72%) rispetto al caso COCO (circa 58%), indicando una maggiore coerenza tra oggetto inserito e contesto della scena.

L'analisi ha tuttavia evidenziato alcune limitazioni. In primo luogo, i modelli di object detection addestrati su COCO mostrano una maggiore efficacia su scene appartenenti allo stesso dominio rispetto a scene acquisite autonomamente. In secondo luogo, l'assenza di un vincolo spaziale nella fase di valutazione non consente di verificare che il bounding box rilevato corrisponda effettivamente alla regione di inserimento dell'oggetto sintetico, rendendo possibile l'identificazione di oggetti presenti nella scena ma diversi da quello generato. Infine, la sperimentazione è stata condotta su un numero limitato di categorie e campioni, in quanto la validazione della correttezza delle immagini generate richiede la valutazione da parte di un operatore umano. Tale vincolo rende onerosa l'estensione dell'analisi a dataset di dimensioni significativamente superiori e suggerisce cautela nella generalizzazione dei risultati.

Gli sviluppi futuri si articolano su più direttrici. Da un lato, si prevede l'introduzione di un criterio di verifica basato su Intersection over Union (IoU), al fine di associare il rilevamento automatico alla specifica regione di inserimento dell'oggetto sintetico e distinguere tra rilevamenti pertinenti e non pertinenti. Dall'altro, sarà opportuno valutare strategie di adattamento o fine-tuning dei modelli di detection su scene acquisite autonomamente, riducendo il gap di dominio osservato rispetto ai dati di addestramento originali. Ulteriori estensioni riguarderanno l'ampliamento del numero di categorie oggetto e la conduzione di sperimentazioni su dataset più estesi.

In particolare, la metodologia basata su modelli di object detection potrà essere evoluta da semplice strumento di analisi statistica a componente attiva della pipeline generativa. Il modulo di riconoscimento potrà infatti essere impiegato come sistema di filtraggio automatico, selezionando esclusivamente le immagini in cui l'oggetto sintetico risulti effettivamente riconoscibile. Un tale meccanismo consentirebbe di automatizzare parzialmente il controllo qualitativo, riducendo l'intervento umano e migliorando l'efficienza complessiva del processo di costruzione del dataset. Un ulteriore sviluppo rilevante consiste nella rimozione completa della dipendenza dall'interfaccia ComfyUI. L'obiettivo è isolare ed estrarre le componenti strettamente necessarie all'esecuzione dei modelli di diffusione, integrandole direttamente nella pipeline sviluppata. Ciò consentirà di ottenere un sistema più leggero, modulare e indipendente, riducendo l'overhead non necessario e facilitando l'integrazione in ambienti *server-side* o produttivi. Questa evoluzione architetturale rappresenta un passo significativo verso la trasformazione del prototipo sviluppato in un'infrastruttura generativa pienamente autonoma.

7.2 Considerazioni sul lavoro di tirocionio

Il lavoro è stato svolto nel rispetto delle tempistiche concordate con l'azienda e ha portato alla realizzazione di un progetto completo, in grado sia di generare immagini sintetiche sia di valutarne automaticamente la qualità. Il sistema sviluppato risulta pienamente predisposto alla condivisione e all'estensione, soddisfacendo gli obiettivi inizialmente prefissati. I risultati e le metodologie presentati costituiscono inoltre la base per un articolo scientifico attualmente in fase di stesura, a conferma della rilevanza e dell'interesse del lavoro svolto anche in ambito di ricerca.

I risultati ottenuti rispettano le aspettative, pur evidenziando margini di miglioramento emersi dalla fase di validazione del dataset.

Il progetto ha rappresentato un'importante opportunità di crescita personale e professionale, permettendomi di approfondire un ambito di forte interesse e di sperimentare direttamente tecnologie innovative e allo stato dell'arte, anche grazie alle risorse e al supporto messi a disposizione dall'azienda.

Durante l'intero periodo di tirocinio, il team aziendale si è dimostrato costantemente disponibile nel fornire supporto tecnico e chiarimenti, favorendo un ambiente di lavoro collaborativo. Inoltre, l'esperienza mi ha consentito di acquisire nuove competenze e metodologie, tra cui la comprensione del funzionamento dei modelli di diffusione, l'utilizzo di librerie specifiche di Deep Learning per l'impiego di modelli pre-addestrati, il lavoro in team, la gestione delle decisioni condivise e il rispetto delle tempistiche di sviluppo.

Ringraziamenti

Questo traguardo è il risultato di una convinzione che mi ha accompagnato in questi anni: affrontare le difficoltà senza arrendersi e considerare i fallimenti non come conclusioni, ma come passaggi indispensabili per proseguire. Questa tesi non rappresenta solo un risultato accademico, ma il frutto di impegno, tentativi, errori e miglioramenti continui. È la dimostrazione che la perseveranza e la costanza, più del talento, fanno la differenza.

Se oggi posso affermarlo con convinzione, è perché ho avuto accanto persone che mi hanno insegnato, con l'esempio, il valore della determinazione. In queste ultime righe desidero quindi ringraziare coloro che mi hanno permesso di raggiungere questo obiettivo e mi sono stati vicini.

Voglio ringraziare mia madre e mio padre per aver reso possibile tutto questo. Con pazienza e fiducia mi hanno insegnato ad andare avanti, a non arrendermi mai, insegnandomi (*o almeno ci hanno provato e continuano tuttora a farlo*) che la felicità vive nelle piccole cose e nel percorso, non solo nel traguardo. Mi sono stati accanto nelle difficoltà e presenti nei successi, anche quando il mio carattere rendeva il cammino meno semplice. Mi hanno dato la libertà di studiare, di provare, di sbagliare e di sperimentare e, ogni volta che la strada si faceva incerta, sono stati per me un faro capace di ricondurmi sempre nella giusta direzione.

Devo a loro tutto e questo obiettivo raggiunto è tanto mio quanto loro.

Voglio ringraziare le mie sorelle, Giulia e Chiara. Questo percorso è anche il risultato della loro presenza costante. Tra noi non è mai mancata una certa competizione, ma è stata proprio quella a diventare uno stimolo continuo a crescere e a dare qualcosa in più.

Voglio ringraziare Martina, che in questi anni mi è sempre stata vicina ed è stata la mia prima sostenitrice, credendo in me prima ancora che lo facessi io.

Ho due cerchie di amici: una composta da musicisti, l'altra no. Diverse, ma entrambe fondamentali. Ringrazio tutti loro per aver condiviso con me esperienze, passioni e momenti che hanno arricchito questo cammino, ben oltre l'ambito accademico.

18 febbraio 2026

Giacomo Torbidoni

Dichiarazione sull'utilizzo di strumenti di supporto alla scrittura

Per la revisione linguistica e il miglioramento della scorrevolezza del testo è stato utilizzato uno strumento di Intelligenza Artificiale generativa (ChatGPT, OpenAI). L'autore dichiara di aver verificato e supervisionato personalmente tutti i contenuti, assumendosi piena responsabilità del lavoro presentato.

Appendici

Le appendici raccolgono alcuni approfondimenti teorici e operativi che, pur non essendo centrali per la comprensione del flusso principale del lavoro, forniscono un supporto utile alla lettura, all'interpretazione dei risultati e alla riproducibilità degli esperimenti descritti nei capitoli precedenti.

A.1 Concetti teorici di base

A.1.1 Tensori

Nel contesto del deep learning, un tensore rappresenta una generalizzazione dei concetti di scalare, vettore e matrice a un numero arbitrario di dimensioni. Le immagini digitali possono essere modellate come tensori tridimensionali di dimensione $H \times W \times C$, dove H e W rappresentano altezza e larghezza dell'immagine, mentre C indica il numero di canali.

Tale rappresentazione risulta fondamentale per l'elaborazione delle immagini nei modelli di intelligenza artificiale generativa, in quanto consente l'applicazione di operazioni matematiche efficienti e parallelizzabili.

A.1.2 Maschere

Le maschere sono immagini binarie o a livelli di grigio utilizzate per identificare regioni di interesse all'interno di una scena. Nel caso dell'inpainting, la maschera definisce le aree in cui il modello è autorizzato a modificare o generare nuovi contenuti, preservando il contesto circostante.

Nel presente lavoro, le maschere sono state impiegate sia in forma statica, derivata dal dataset, sia in forma dinamica, costruita automaticamente in fase di sperimentazione per migliorare il realismo prospettico degli oggetti inseriti.

A.1.3 Priors

Nei modelli generativi, il termine *prior* indica una distribuzione probabilistica che codifica assunzioni preliminari sui dati. I prior influenzano il processo di generazione guidando il modello verso soluzioni coerenti con le statistiche apprese durante l'addestramento.

Nel contesto delle immagini sintetiche, i prior contribuiscono a mantenere coerenza semantica e visiva, evitando generazioni implausibili o incoerenti con la scena di riferimento.

A.2 Tecniche di addestramento e stabilità

A.2.1 Mode Collapse

Il *mode collapse* è un fenomeno noto nei modelli generativi, in cui il modello tende a produrre un numero limitato di variazioni, ignorando ampie porzioni della distribuzione dei dati. Questo comportamento compromette la diversità delle generazioni e riduce l'utilità del dataset sintetico prodotto.

La presenza di tecniche di regularizzazione, un'adeguata scelta dei parametri e l'uso di maschere contestuali contribuiscono a mitigare tale problema.

A.3 Prompt e configurazioni operative

A.3.1 Struttura dei prompt testuali

I prompt testuali utilizzati nel processo di generazione sono stati progettati per bilanciare la descrizione semantica dell'oggetto da inserire, la coerenza con il contesto della scena e il controllo stilistico del risultato finale. In particolare, il sistema di generazione adotta una distinzione esplicita tra *prompt positivi* e *prompt negativi*, entrambi fondamentali per guidare efficacemente il comportamento del modello generativo.

Prompt positivi

I prompt positivi definiscono le caratteristiche desiderate dell'immagine generata e specificano ciò che il modello dovrebbe includere nel risultato finale. Essi forniscono una descrizione semantica dell'oggetto da inserire, insieme a indicazioni contestuali e stilistiche coerenti con la scena di riferimento.

Tipicamente, un prompt positivo include:

- una descrizione esplicita dell'oggetto da generare;
- riferimenti al contesto spaziale e visivo della scena;
- indicazioni di stile, qualità e realismo visivo.

Prompt negativi

I prompt negativi specificano gli elementi che il modello dovrebbe evitare durante il processo di generazione. Essi rappresentano uno strumento di controllo fondamentale per limitare comportamenti indesiderati tipici dei modelli generativi, come la produzione di artefatti o incoerenze visive.

In particolare, i prompt negativi sono utilizzati per:

- ridurre la comparsa di artefatti visivi e rumore;
- evitare oggetti duplicati o semanticamente errati;
- limitare stili grafici non realistici o non desiderati;
- prevenire distorsioni prospettiche o geometriche.

Bibliografia

[1] Benlisquare. *X-Y plot of algorithmically-generated AI art of European-style castle in Japan demonstrating DDIM diffusion steps.* https://commons.wikimedia.org/wiki/File:X-Y_plot_of_algorithmically-generated_AI_art_of_European-style_castle_in_Japan_demonstrating_DDIM_diffusion_steps.png. Immagine rilasciata sotto licenza Creative Commons Attribution-ShareAlike 4.0. Nov. 2022.

[2] GeeksforGeeks. *U-Net Architecture Explained.* https://www.geeksforgeeks.org/machine-learning/u-net-architecture-explained/. Accessed: 2026-02-02. 2023.

[3] Ian Goodfellow et al. «Generative Adversarial Nets». In: *arXiv preprint arXiv:1406.2661* (2014). URL: https://arxiv.org/abs/1406.2661.

[4] Jonathan Ho, Ajay Jain e Pieter Abbeel. «Denoising Diffusion Probabilistic Models». In: *arXiv preprint arXiv:2006.11239* (2020). URL: https://arxiv.org/abs/2006.11239.

[5] Alex Krizhevsky, Ilya Sutskever e Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems* 25 (2012). URL: https://arxiv.org/abs/1207.0580.

[6] Chaojie Mao et al. «ACE++: Instruction-Based Image Creation and Editing via Context-Aware Content Filling». In: *arXiv preprint arXiv:2501.02487* (2025).

[7] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». In: *arXiv preprint arXiv:1506.02640* (2015).

[8] Robin Rombach et al. «High-Resolution Image Synthesis with Latent Diffusion Models». In: *arXiv preprint arXiv:2112.10752* (2021). URL: https://arxiv.org/abs/2112.10752.

[9] Olaf Ronneberger, Philipp Fischer e Thomas Brox. «U-Net: Convolutional Networks for Biomedical Image Segmentation». In: *arXiv preprint arXiv:1505.04597* (2015). URL: https://arxiv.org/abs/1505.04597.

[10] Connor Shorten e Taghi M. Khoshgoftaar. «A Survey on Image Data Augmentation for Deep Learning». In: *Journal of Big Data* 6.1 (2019), pp. 1–48. DOI: 10.1186/s40537-019-0197-0.

[11] Jascha Sohl-Dickstein et al. «Deep Unsupervised Learning using Nonequilibrium Thermodynamics». In: *arXiv preprint arXiv:1503.03585* (2015). URL: https://arxiv.org/abs/1503.03585.

[12] Candi Zheng, Yuan Lan e Yang Wang. «LanPaint: Training-Free Diffusion Inpainting with Asymptotically Exact and Fast Conditional Sampling». In: *Transactions on Machine Learning Research* (2025). ISSN: 2835-8856. URL: https://openreview.net/forum?id=JPC8JyOUSW.