UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e validazione di una rete generativa avversaria basata sui transformer per attività di Unpaired Image-to-Image Translation

Design and validation of a transformer-based generative adversarial network for Unpaired Image-to-Image Translation activities

Relatore

Prof. Domenico Ursino

Candidato

Andrea Maranesi

Correlatore

Dott. Luca Virgili

ANNO ACCADEMICO 2022-2023

Sommario

Nel campo del deep learning, l'Unpaired Image-to-Image Translation rappresenta un task dove si cerca di trasformare un'immagine da un dominio *A* a uno *B* senza avere i dati accoppiati. Un traguardo fondamentale è stato il lavoro *CycleGAN* del 2017, dove si propone un generatore basato su blocchi della ResNet, e una rete avversaria basata su layer convoluzionali. Tuttavia, in quel lavoro e in altri successivi, nella rete avversaria non vengono introdotti i meccanismi di attenzione tipici dell'articolo *Attention is All You Need*.

Il nostro studio propone, così, un nuovo approccio, ovvero lo sviluppo di un discriminatore basato sui Transformer. Il nostro modello, che abbiamo chiamato T-PatchGAN, sfrutta l'attenzione locale sui pixel e affronta la sfida di bilanciare le capacità del discriminatore rispetto al generatore. Per superare alcune limitazioni della T-PatchGAN nell'analisi delle texture globali, abbiamo introdotto la TGlobal-PatchGAN, che abbina l'estrazione di feature locali e globali, migliorando la qualità dei risultati del generatore.

Keyword: Generative Adversarial Network, Unpaired Image-to-Image, CNN, Transformer, CycleGAN, Torch

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Con l'avvento delle reti neurali e l'evoluzione del deep learning a cui abbiamo assistito negli ultimi anni, un dominio specifico delle reti generative, denominato Unpaired Image-to-Image Translation, sta emergendo con nuove soluzioni. In particolare, tale dominio si occupa di partire da due domini, A e B, per allenare una rete, detta generatore, a trasformare le immagini da un dominio all'altro senza avere i dati accoppiati, ovvero esempi di traslazioni dell'immagine del dominio A in quello B, e viceversa. Come nel contesto più ampio delle Generative Adversarial Network, è cruciale anche l'uso di una rete avversaria (discriminatore) per migliorare il realismo dell'immagine trasformata. Questo richiede un equilibrio delicato tra il generatore e la rete avversaria, presentando una sfida notevole. Un lavoro importante in questo ambito è *CycleGAN* del 2017, che è stato uno dei capisaldi in questo settore. L'articolo *Attention is All You Need*, che introduce invece il meccanismo della self-attention dei Transformer, non era ancora apparso quando iniziavano le prime importanti ricerche sull'Unpaired Image-to-Image Translation. Di conseguenza, i primi generatori e reti avversarie, come quelli di CycleGAN, si basavano su idee preesistenti, quali la ResNet, con blocchi convoluzionali alla base dei generatori. Gli autori di CycleGAN hanno introdotto PatchGAN come rete avversaria, efficace ma ancora basata su layer convoluzionali, senza meccanismi di attenzione sui patch d'immagine. Negli ultimi anni, tuttavia, sono state proposte varianti basate sull'attenzione, tra cui un importante articolo che introduce l'architettura *U-GAT-IT*, dove generatore e discriminatore si basano su meccanismi di attenzione per selezionare le feature più importanti dei blocchi convoluzionali, senza, però, introdurre il calcolo matriciale per l'attenzione proposto in *Attention is All You Need*. Un passo avanti importante è stato fatto dagli autori di *UVCGAN*, che hanno introdotto un generatore basato sulla U-NET, la cui bottleneck (parte finale dell'encoder) è un Transformer Encoder. La rete avversaria rimane la PatchGAN, e nel campo mancano ancora approcci in cui anche il discriminatore sia basato sui Transformer. L'obiettivo è, quindi, quello di proporre una rete avversaria basata sui layer dei Transformer, in grado di guidare il training in modo efficiente ed efficace, portando a risultati qualitativamente migliori a parità di epoche, superando, così, gli approcci classici ancora basati su reti puramente convoluzionali.

Nel nostro lavoro, esaminiamo prima gli approcci esistenti nello stato dell'arte per poi sviluppare un'architettura di rete avversaria basata sui meccanismi dei Transformer e dei Vision Transformer. L'obiettivo è migliorare l'analisi delle feature nelle immagini per potenziare il generatore utilizzato. Inizialmente, partendo dall'idea della PatchGAN, passiamo le feature estratte da dei layer convoluzionali in un Transformer Encoder che lavora su patch locali, classificandoli come 'Fake' (0) o 'Real' (1), creando, così, la T-PatchGAN. Questo modello enfatizza l'attenzione sui singoli pixel dei patch, e non vede mai insieme tutti i patch come un

classico Vision Transformer, evitando che la rete avversaria diventi troppo potente rispetto al generatore. Tuttavia, per affrontare la limitazione dell'incapacità di estrarre feature globali della T-PatchGAN, introdurremo la TGlobal-PatchGAN. Quest'ultima non solo sfrutta la T-PatchGAN per le feature locali ma, utilizzando un approccio convoluzionale parallelo, estrae anche texture globali. Questo approccio ibrido migliora i risultati, superando la PatchGAN sia in termini di efficacia computazionale che di qualità dei risultati del generatore.

La tesi si articola in sette capitoli, strutturati come di seguito specificato:

- Nel Capitolo 1 vengono introdotti i concetti fondamentali dei Transformer e Vision Transformer, seguiti da un approfondimento sull'Image-to-Image Translation, ponendo l'attenzione sull'articolo della CycleGAN.
- Nel Capitolo 2 si analizzano le reti CycleGAN e PatchGAN, approfondendo le recenti innovazioni nel campo, incluse U-GAT-IT e UVCGAN.
- Nel Capitolo 3 presentiamo la T-PatchGAN, una rete avversaria che incorpora il meccanismo dei Vision Transformer per l'estrazione di feature locali.
- Nel Capitolo 4 descriviamo l'implementazione Python della rete, includendo anche i dettagli tecnici per il training.
- Nel Capitolo 5 esponiamo i risultati degli esperimenti condotti utilizzando il generatore UVCGAN, la T-PatchGAN e la PatchGAN, su tre diversi dataset.
- Nel Capitolo 6 superiamo le limitazioni della T-PatchGAN con la TGlobal-PatchGAN, che si dimostra una valida alternativa alla PatchGAN.
- Nel Capitolo 7 discutiamo in merito a sviluppi futuri di queste idee di ricerca partendo dai risultati ottenuti con la TGlobal-PatchGAN.

CAPITOLO 1

Vision Transformer nell'Image-to-Image Translation

In questo capitolo iniziale, analizzeremo la tematica emergente dell'Image-to-Image Translation, esplorando le sue applicazioni più rilevanti nel campo del deep learning. Considerata la recente diffusione delle architetture basate su transformer, iniziata con la pubblicazione del paper "Attention is All You Need" e proseguita con l'implementazione di Vision Transformer per la classificazione di immagini, cerheremo di illustrare le componenti principali di queste reti.

In seguito, approfondiremo il sotto-dominio specifico dell'Unpaired Image-to-Image Translation. In questo contesto, la sfida principale risiede nell'assenza di corrispondenze dirette tra dati di input e output. Discuteremo quindi le complessità e le sfide inerenti a questo particolare ramo dell'Image-to-Image Translation.

1.1 Cosa sono i Vision Transformer

I Vision Transformer (ViT) sono una classe di modelli di deep learning che hanno recentemente guadagnato popolarità nel campo della computer vision. Contrariamente ai modelli tradizionali basati su reti neurali convoluzionali (CNN), i ViT si appoggiano alla potenza dei transformer, una famiglia di modelli introdotta nell'articolo "Attention Is All You Need" [Vaswani *et al.*, 2017].

1.1.1 Convolutional Neural Networks (CNNs)

Le CNN sono composte da layer convoluzionali, layer di pooling e layer fully connected. Ogni layer convoluzionale è specializzato per identificare particolari aspetti dell'immagine, come bordi, colori o forme. I layer di pooling riducono la dimensione spaziale dell'immagine, preservando le caratteristiche più importanti. Infine, i layer fully connected riducono la dimensione spaziale a un output, ad esempio per effettuare la classificazione (Figura 1.1).

1.1.2 Funzionamento dei Vision Transformer

Architettura del Transformer

A differenza delle *CNN (Convolutional Neural Network)*, i *transformer* utilizzano meccanismi di *self-attention* per pesare l'importanza di diverse parti dell'input, sia esso una sequenza di parole o un'immagine. Questo permette al modello di focalizzarsi su aree rilevanti dell'immagine, considerando il contesto globale piuttosto che solo una porzione locale. In pratica, mentre un layer convoluzionale potrebbe essere addestrato per riconoscere un occhio

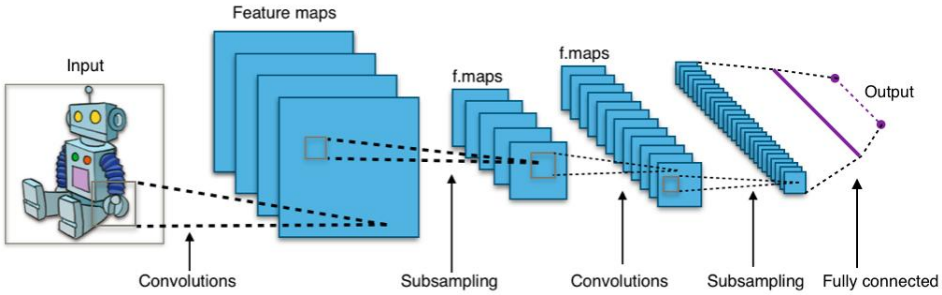

Figura 1.1: Esempio Architettura CNN

o una bocca, un layer di *self-attention* in un transformer potrebbe imparare a focalizzarsi sull'interazione tra occhi e bocca per riconoscere un'espressione facciale (Figura 1.2).

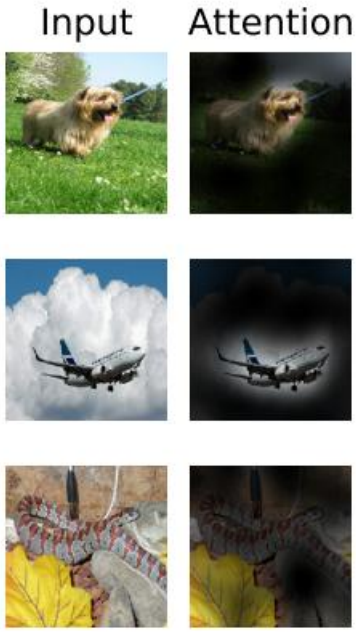

Figura 1.2: Esempio di *Self-Attention* di un Vision Transformer

L'architettura generica, evidenziata in Figura 1.3, e proposta dall'articolo *"Attention is All You Need"*, consiste principalmente di una fase di *encoding*, per l'input, e una di *decoding*, dove l'output e ciò che è stato appreso dall'encoder si uniscono per produrre la previsione finale. L'input e l'output sono vettori in uno spazio latente, ai quali viene aggiunto un *positional encoding* non allenabile. Il *positional encoding* viene aggiunto al vettore di input per fornire informazioni sulla posizione del token all'interno della sequenza.

Successivamente, si entra nel modulo *Multi-Head Attention*, dove l'input viene diviso in tre vettori: *Query*, *Key*, *Value*. Questi sono utilizzati nella computazione matriciale dell'attenzione, motivo per il quale il transformer ha un costo computazionale di $O(L^2 \times d)$, dove L è la lunghezza dell'input e d è la dimensione latente.

La matrice dell'attenzione viene generata moltiplicando la *Query* (Q) con la *Key* (K). La formula completa per il calcolo dell'attenzione è

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

dove d_k è la dimensione del vettore *Key*. La divisione per $\sqrt{d_k}$ è una normalizzazione che aiuta a stabilizzare i valori all'interno della funzione softmax.

Una volta calcolata la matrice, questa viene moltiplicata per il vettore *Value* (V) per ottenere il vettore di output dell'attenzione (Figura 1.4). Questo processo consente al modello di assegnare pesi differenti ai diversi token nell'input, in base alle loro relazioni globali.

Per aumentare la capacità del modello di analizzare diverse feature del vettore di partenza, nel modulo **Multi-Head Attention**, l'attenzione può essere calcolata più volte in parallelo, con diverse rappresentazioni lineari dei vettori *Query*, *Key* e *Value*. Formalmente, ciò è rappresentato come:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{1.1.1}$$

dove ogni *head* è calcolata come:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{1.1.2}$$

Dopo il calcolo dell'attenzione multi-head, l'output passa attraverso una rete feed-forward. Questa rete consiste di due livelli di trasformazioni lineari con una funzione di attivazione ReLU tra di loro:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{1.1.3}$$

Infine, l'output di ogni sotto-modulo viene sempre sommato all'input originario, e normalizzato tramite la funzione *LayerNorm*.

Gli encoder e decoder possono essere impilati in sequenza, noto come meccanismo di *stacking*, per permettere al modello di apprendere pattern più complessi.

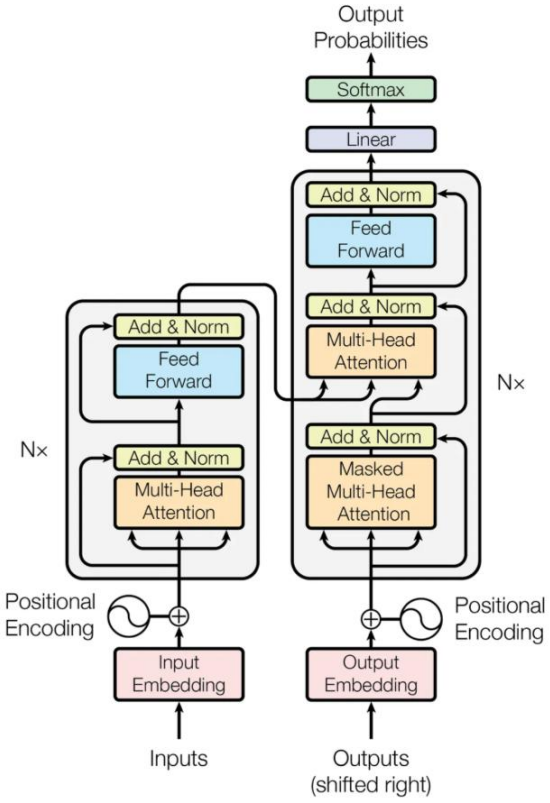

Figura 1.3: Architettura di un Transformer

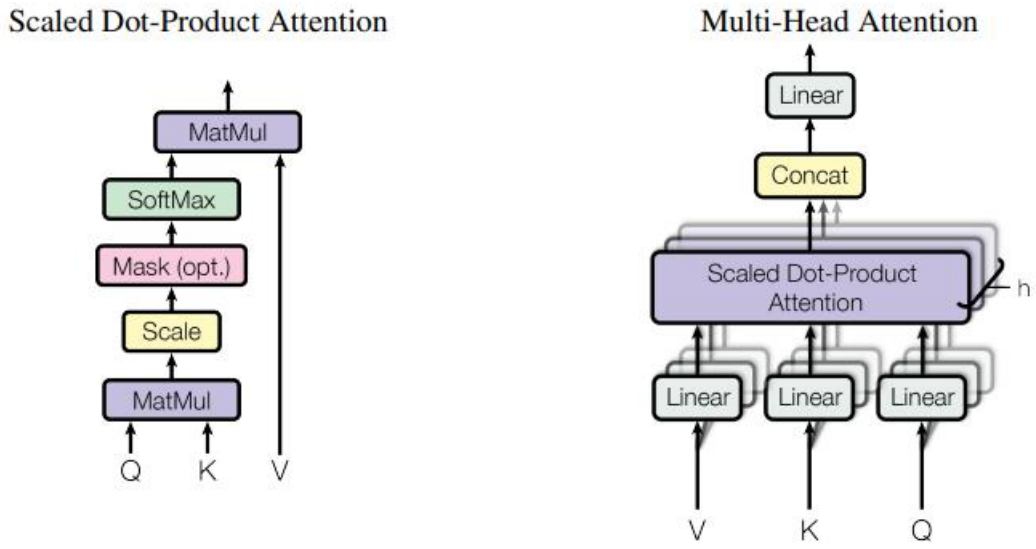

Figura 1.4: Scaled Dot-Product Attention

Vision Transformer (ViT)

Gli autori del Vision Transformer [Dosovitskiy *et al.*, 2020] hanno proposto una variante che è in grado di elaborare immagini segmentandole in patch non sovrapposti. Ogni patch viene quindi appiattito in un vettore unidimensionale e passato attraverso i layer di attenzione. Questa architettura, illustrata in Figura 1.5, è in molti aspetti simile all'architettura originale del Transformer, con la differenza che l'input è costituito da patch di un'immagine.

Similmente al token $[class]$ utilizzato in BERT, un *embedding* apprendibile viene anteposto alla sequenza dei vettori di patch, prima di entrare nel Multi-Head Attention. Lo stato di questo embedding serve, all'uscita dell'encoder del Transformer, come rappresentazione globale dell'immagine.

Per conservare le informazioni sulla posizione dei patch all'interno dell'immagine, vengono aggiunti degli *embedding* di posizione **allenabili** - rispetto la versione proposta originariamente - ai vettori di patch.

Inoltre, in questa variante, i layer di normalizzazione sono anteposti a quelli di *Multi-Head Attention* e *MLP*.

Nel paper, è utilizzata la funzione di attivazione GELU (Gaussian Error Linear Unit), nel modulo *MLP* (analogo al modulo *Feed Forward*), definita come segue:

$$\text{GELU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

La GELU è una funzione *smooth*, ovvero è differenziabile in modo continuo, a differenza della ReLU, che presenta una discontinuità allo zero, ciò può favorire l'ottimizzazione basata sul gradiente durante la fase di addestramento.

Oltre al Vision Transformer standard, esistono diverse varianti che implementano approcci innovativi. Ad esempio, il **Convolutional Compact Transformer** [Hassani *et al.*, 2021] diverge dalla struttura tradizionale del ViT, optando per un meccanismo di *Sequential Pooling* al posto del token $[class]$, come illustrato in Figura 1.6. Un'altra variante è la **CrossViT** [Chen *et al.*, 2021a], che introduce meccanismi addizionali per ottimizzare il modello in contesti specifici.

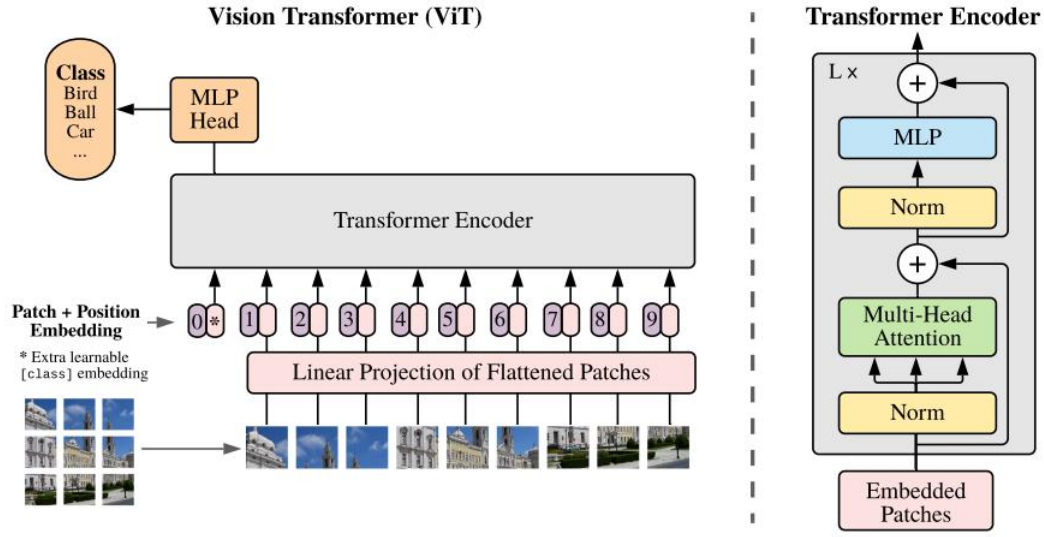

Figura 1.5: Architettura di un Vision Transformer

Questi sono solo alcuni esempi che evidenziano la crescente diversità e specializzazione nel campo dei Vision Transformer.

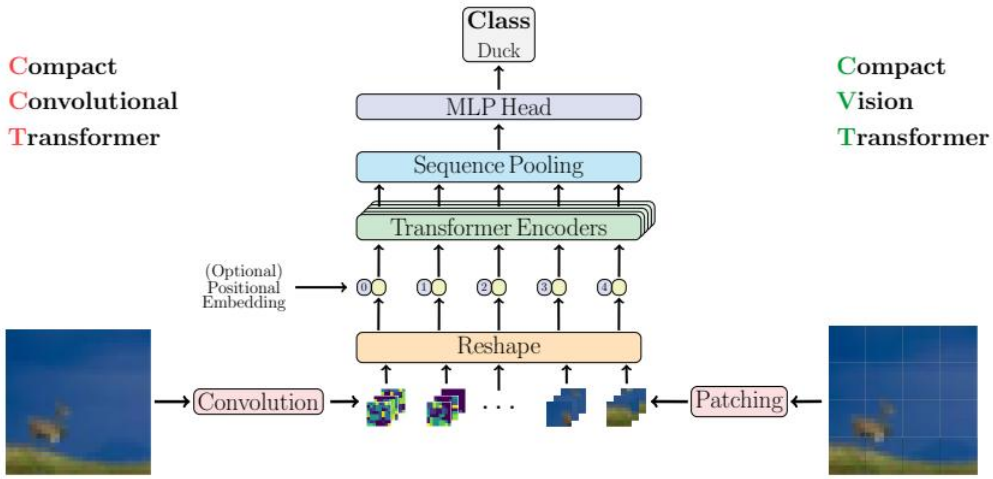

Figura 1.6: Convolutional Compact Transformer

1.1.3 Vantaggi e svantaggi dei Vision Transformer

I Vision Transformer offrono diversi vantaggi rispetto ai metodi tradizionali; tra questi citiamo:

- **Flessibilità**: Essendo basati sui transformer, i ViT sono intrinsecamente più flessibili e possono gestire un'ampia varietà di input. A differenza delle CNN, che richiedono generalmente un input di dimensione fissa, i transformer possono accettare input di dimensioni variabili senza la necessità di modificare l'architettura del modello.

- **Interpretabilità**: Il meccanismo di *attention* permette di generare una sorta di "mappa di calore" che indica quali parti dell'immagine sono state ritenute più importanti durante la classificazione. Questo rende i ViT più interpretabili e può aiutare a comprendere meglio il processo decisionale del modello.

Tuttavia, i Vision Transformer presentano anche alcuni svantaggi; tra questi citiamo:

- **Costo computazionale**: I transformer sono notoriamente onerosi in termini di risorse computazionali. L'uso del meccanismo di *attention* richiede una grande quantità di memoria e potenza di calcolo, soprattutto quando si lavora con immagini ad alta risoluzione o grandi set di dati, come descritto nel paragrafo precedente.
- **Necessità di grandi dataset**: I ViT sono modelli potenti ma esigenti in termini di dati. Per raggiungere la loro piena efficacia, richiedono l'addestramento su un ampio set di dati. Ciò può rappresentare una sfida quando si hanno a disposizione solo dati limitati o quando si lavora in ambiti specifici dove la raccolta dei dati è difficile.

Come detto, uno degli aspetti più critici nell'utilizzo dei Vision Transformer è il costo computazionale. I meccanismi di *attention* richiedono il calcolo di pesi di attenzione per ogni coppia di patch dell'immagine, il che può diventare rapidamente proibitivo per immagini di grandi dimensioni. Questo è particolarmente vero nei modelli di transformer più grandi, che possono avere milioni o addirittura miliardi di parametri.

1.1.4 Applicazioni pratiche

Grazie alla loro versatilità e capacità di catturare relazioni complesse nei dati attraverso il meccanismo della *self-attention*, i Vision Transformer si sono rapidamente affermati come uno degli strumenti più potenti nel campo dell'NLP, ma anche in altri ambiti. Infatti, l'architettura dei Transformer, con opportune modifiche, è stata applicati con successo in una varietà di contesti, come la segmentazione semantica, di cui un esempio è l'archiettura [Chen *et al.*, 2021b] mostrata in Figura 1.7, e il riconoscimento di oggetti [Carion *et al.*, 2020]. Recentemente, i Transformer sono stati utilizzati anche per affrontare problemi di forecasting di serie temporali [Xue *et al.*, 2023].

1.2 Introduzione all'Image-to-Image Translation

1.2.1 Panoramica dell'Image-to-Image Translation

L'Image-to-Image Translation è un dominio della computer vision che si focalizza sulla trasformazione di un'immagine di input in un'immagine di output coerente. Una delle architetture più influenti in questo ambito sono le Generative Adversarial Networks (GANs). In particolare, il modello Pix2Pix [Isola *et al.*, 2017] e CycleGAN [Zhu *et al.*, 2017], sono stati un passo fondamentale per stabilire nuovi standard nella disciplina.

1.2.2 Applicazioni del Deep Learning nell'Image-to-Image Translation

Tra le applicazioni dell'Image-to-Image Translation vi sono le seguenti:

1. **Super-risoluzione:** una delle applicazioni emergenti dell'Image-to-Image Translation è nel campo della Super-risoluzione. Uno degli articoli importanti che esplora questa applicazione è "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network" [Ledig *et al.*, 2017], dove si cerca di ricostruire dettagli di bassa qualità dell'immagine (Figura 1.8).

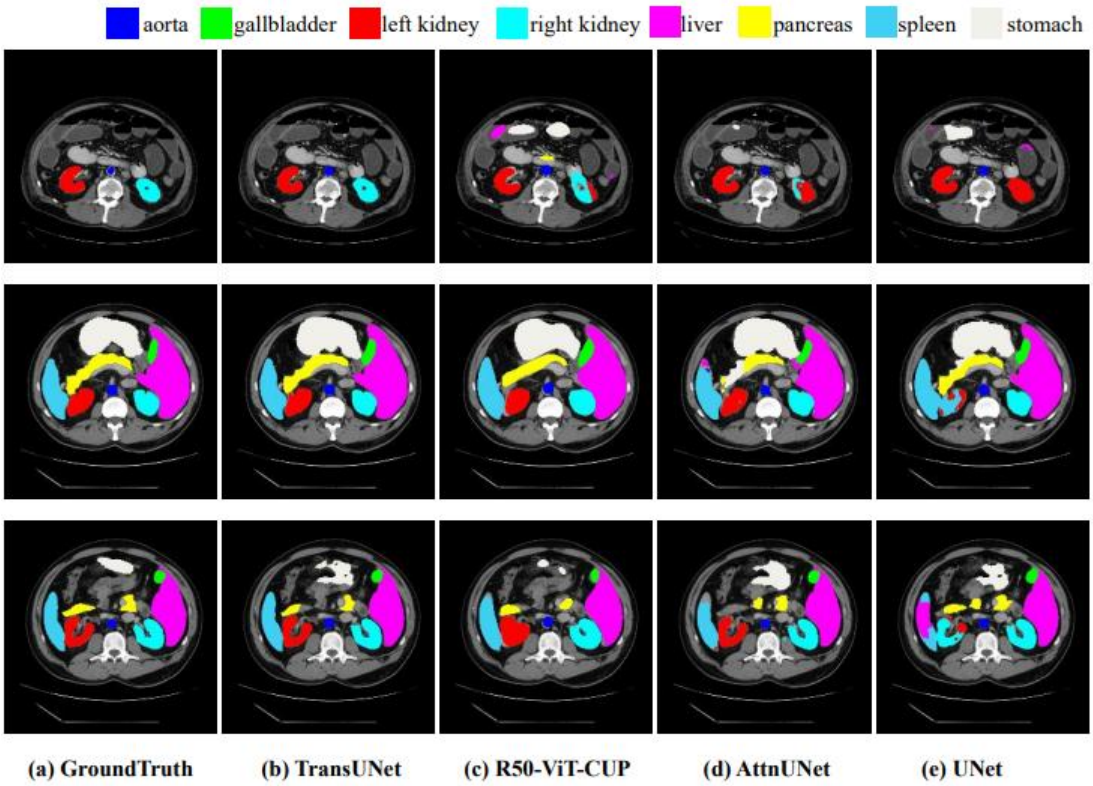

Figura 1.7: TransUNet: Transformer per la segmentazione semantica

2. **Colorizzazione di immagini in bianco e nero:** un altro utilizzo notevole del Deep Learning in questo campo è la colorizzazione di immagini in bianco e nero. L'articolo "Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification" [Iizuka e Edgar Simo-Serra, 2016] presenta un approccio per aggiungere colore a immagini in scala di grigi utilizzando reti neurali (Figura 1.9).

3. **Stilizzazione dell'immagine:** l'Image-to-Image Translation è anche applicata nella stilizzazione di immagini, dove lo stile di un'immagine viene trasformato pur mantenendo il suo contenuto originale. Già qualche anno fa, veniva proposto l'articolo "A Neural Algorithm of Artistic Style" [Gatys *et al.*, 2015], che propone un algoritmo basato su CNN per applicare stili artistici alle immagini (Figura 1.10).

1.2.3 I Transformer nell'Image-to-Image Translation

La tendenza principale, in questo settore, è quella delle cosidette **architetture ibride**, capaci di combinare il potere di catturare feature delle reti convoluzionali e quello dei meccanismi di attenzione dei transformer. Questi modelli ibridi cercano di ottenere il meglio da entrambi i mondi: l'efficienza computazionale delle reti convoluzionali e la capacità di catturare dipendenze globali, sui patch d'immagine, dei transformer.

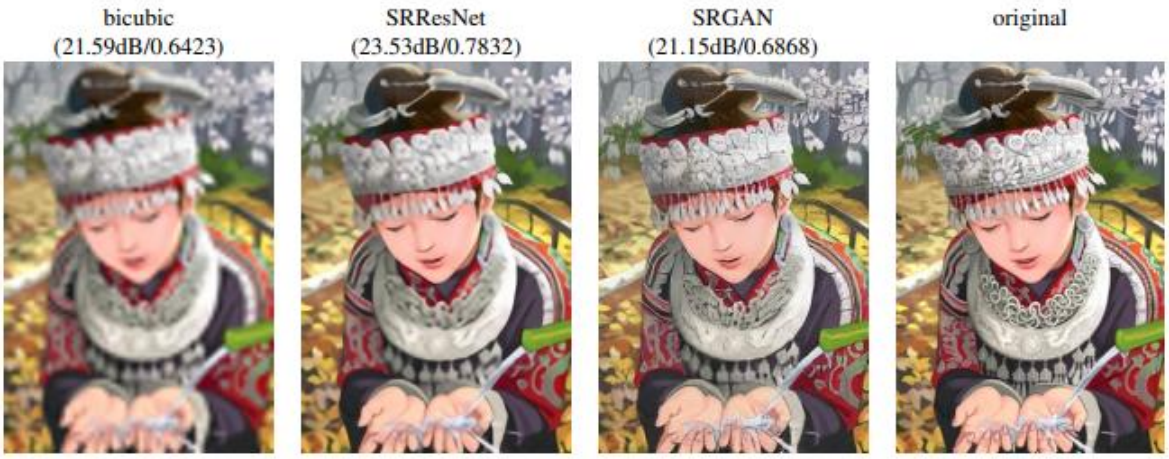

Figura 1.8: Esempio di Super-Risoluzione

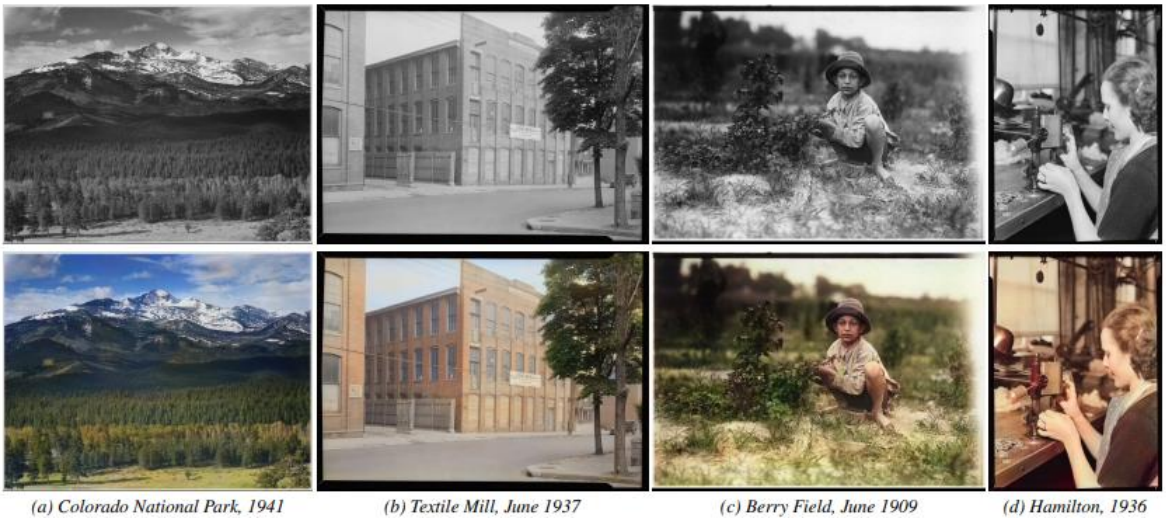

Figura 1.9: Immagini da bianco in nero a colori

1.3 Unpaired Image-to-Image Translation

1.3.1 Definizione e Sfide dell'Unpaired Image-to-Image Translation

L'Unpaired Image-to-Image Translation è un sottoinsieme del problema più generale dell'Image-to-Image Translation, che non richiede una corrispondenza uno-a-uno tra le immagini di input e output durante la fase di addestramento. In altre parole, il modello impara a tradurre immagini tra due dominii senza aver bisogno di esempi "abbinati" che mostrino la stessa scena o lo stesso oggetto nei due dominii. Ciò rende la tecnica molto versatile, ma introduce anche diverse sfide. Ad esempio, il modello deve essere in grado di catturare le distribuzioni di probabilità sottostanti dei due domini in modo accurato. Un articolo chiave in questo campo è "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" [Zhu *et al.*, 2017], che introduce il concetto di consistenza ciclica per affrontare queste sfide.

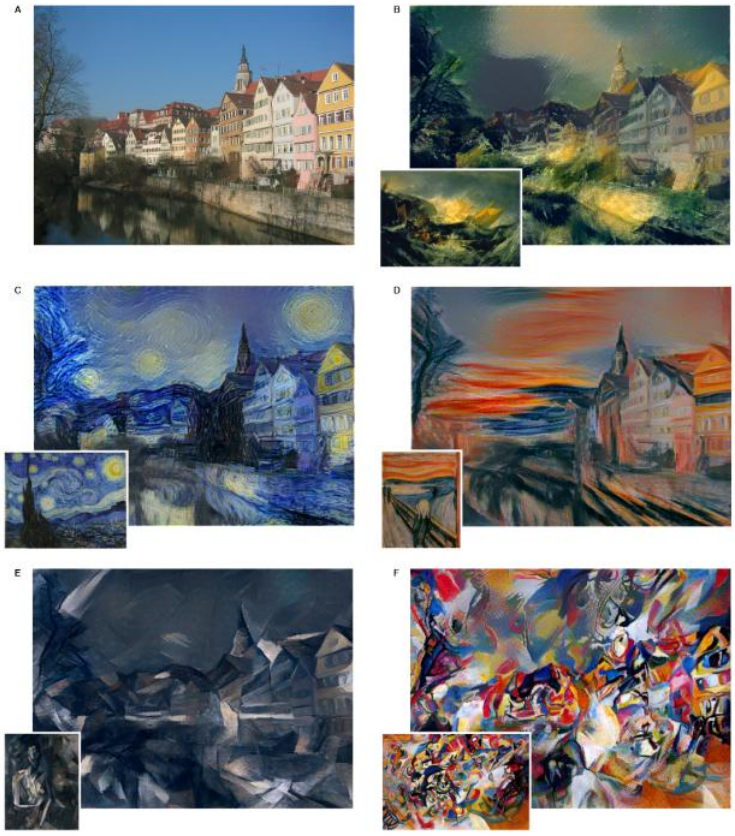

Figura 1.10: Dalla foto all'immagine artistica

1.3.2 Esempi di Applicazioni

Una delle applicazioni mediche significative nell'ambito dell'Unpaired Image-to-Image Translation riguarda l'adattamento di questo dominio per l'interpretazione delle immagini radiologiche. L'articolo *"TUNA-Net: Task-driven Unsupervised Domain Adaptation Network for Disease Recognition"* [Tang *et al.*, 2019b], ad esempio, propone un metodo innovativo che che facilita la diagnosi in ambito pediatrico. Il sistema converte immagini a raggi X del torace da adulti a corrispondenti immagini pediatriche, sintetizzando dati preziosi per l'apprendimento automatico nel riconoscimento di diverse patologie, un'area dove i dati sono spesso limitati (Figura **??**).

Un altro ambito di applicazione altrettanto interessante è la segmentazione di immagini stradali senza la necessità di dati accoppiati. Vari lavori, incluso lo stesso [Zhu *et al.*, 2017], offrono soluzioni efficaci, come dimostrato nella Figura 1.11. Inoltre, studi più recenti presentano tecniche efficienti per eliminare effetti atmosferici come la pioggia da immagini di paesaggi, come descritto da [Wei *et al.*, 2023] (Figura 1.12).

La versatilità delle tecniche di Unpaired Image-to-Image Translation si manifesta in un'ampia gamma di applicazioni, e la letteratura in questo campo è in rapida espansione, con nuovi sviluppi che continuano a estendere le frontiere di ciò che è possibile fare.

1.3.3 La Cyclegan Loss e l'Identity Loss

Nell'ambito dell'unpaired image-to-image, l'articolo di [Zhu *et al.*, 2017] propone l'utilizzo di due generatori e due discriminatori, uno per ciascun dominio, legati tra loro dalla funzione chiamata CycleGAN Loss. Dati i domini di partenza X e Y, questa funzione, oltre alla classica

Figura 1.11: Segmentazioni di immagini urbane

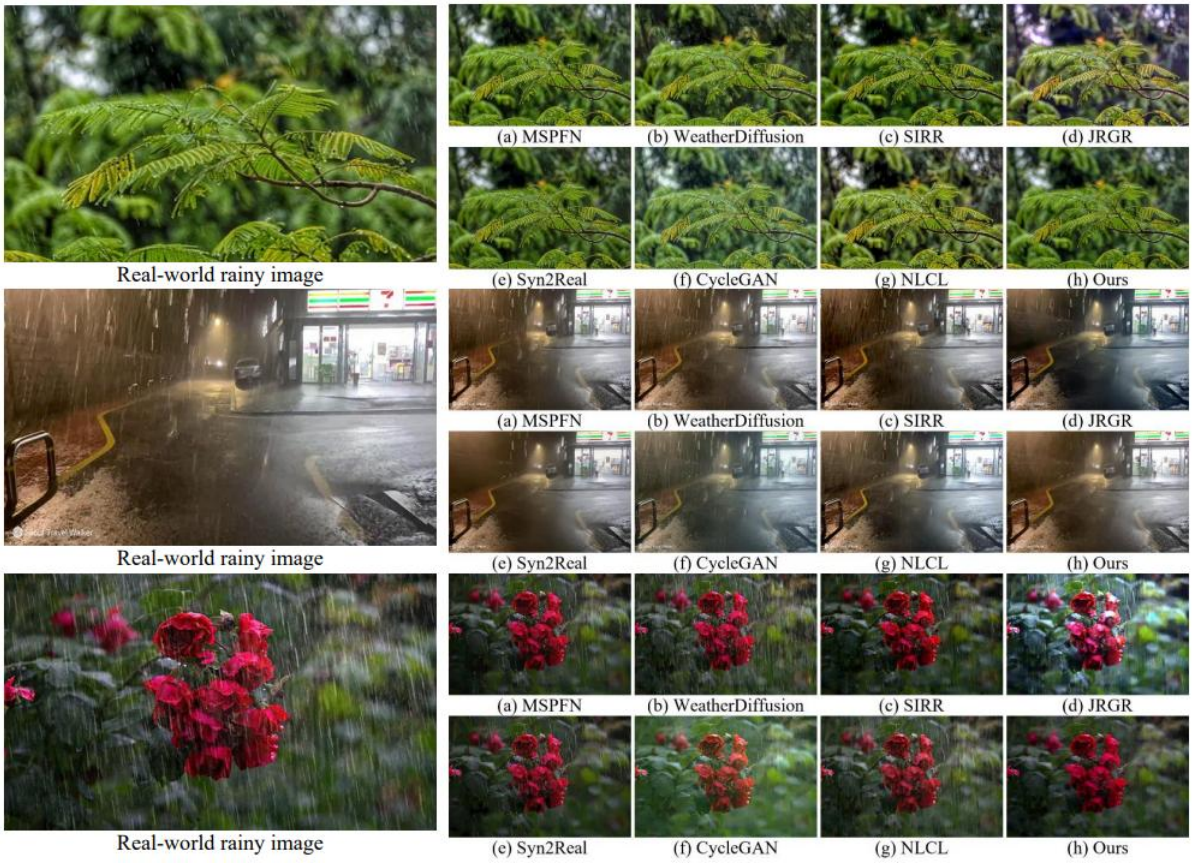

Figura 1.12: Rimozione di pioggia da paesaggi

loss che esiste tra generatore e discriminatore, dove il discriminatore cerca di rendere più verosimile l'immagine generata, introduce questo doppio legame (Figura 1.13) tra i generatori $G : X \to Y$, e $F : Y \to X$. Matematicamente, la CycleGAN Loss è formulata come:

$$\mathcal{L}(G, F, D_x, D_y) = \mathcal{L}_{\text{GAN}}(G, D_y, x, y) + \mathcal{L}_{\text{GAN}}(F, D_x, y, x) + \lambda \mathcal{L}_{\text{cyc}}(G, F) \tag{1.3.1}$$

Dove $\mathcal{L}_{\text{GAN}}$ e $\mathcal{L}_{\text{cyc}}$ rappresentano rispettivamente la GAN Loss e la Cycle Consistency Loss. Per il generatore G, la $\mathcal{L}_{\text{GAN}}$ loss può essere rappresentata come

$$\min_{D_Y} \max_{G} L_{\text{GAN}}(G, D_y, x, y) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \ell_{\text{GAN}}(D_y(y), 1) + \mathbb{E}_{x \sim p_{\text{data}}(x)} \ell_{\text{GAN}}(D_y(G(x)), 0) \tag{1.3.2}$$

dove ℓ_{GAN} è generalmente una funzione di loss $L1$,$L2$, Binary Cross Entropy (BCE). La *Cycle Consistency Loss*, invece, garantisce che un'immagine, quando tradotta da un dominio all'altro e poi ritradotta al dominio originale, dovrebbe apparire simile all'immagine originale. Matematicamente, per il generatore G e F, può essere espressa come:

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \ell_{cyc}(F(G(x)), x) + \mathbb{E}_{x \sim p_{\text{data}}(y)} \ell_{cyc}(G(F(y)), y) \tag{1.3.3}$$

Nel paper originario [Zhu *et al.*, 2017] si ha che ℓ_{cyc} = L1 Loss. Oltre a queste, in alcuni scenari dove il mantenimento dei colori e delle texture è importante (Figura 1.14), viene introdotta l'*Identity Loss*. Questa funzione garantisce che se si prende un'immagine dal dominio Y e si passa attraverso il generatore $G : X \to Y$, l'output dovrebbe essere identico all'immagine di input. Ciò aiuta a preservare le informazioni chiave dell'immagine durante la traduzione. Matematicamente, per il generatore G, è definita come segue:

$$\mathcal{L}_{idt}(G) = \mathbb{E}_{y \sim p_{\text{data}}(y)} \ell_{idt}(G(y), y) \tag{1.3.4}$$

Nel paper originario [Zhu *et al.*, 2017] si ha che ℓ_{idt} = L1 Loss

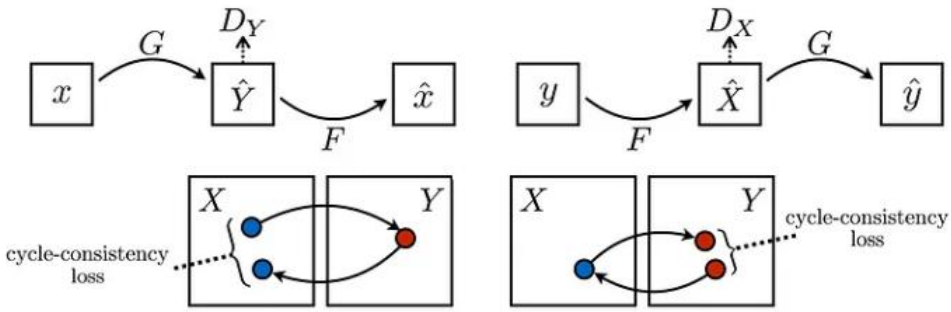

Figura 1.13: Illustrazione della CycleGAN Loss

1.3.4 Equilibrio tra Generatore e Rete Avversaria

Un elemento chiave nella formazione delle Generative Adversarial Networks (GAN) è l'importanza di stabilire un giusto equilibrio tra il generatore e il discriminatore, noto anche come rete avversaria. È cruciale che il generatore non superi in velocità di apprendimento il discriminatore. In particolare, nell'ambito dell'Unpaired Image-to-Image Translation, è di fondamentale importanza che il discriminatore sia in grado di modulare efficacemente l'apprendimento del generatore. Per questa ragione l'articolo che introduce PatchGAN [Demir e Unal, 2018], che verrà esaminato nei capitoli successivi, elabora una strategia specifica, focalizzata sul design della rete avversaria, proprio per preservare tale bilanciamento.

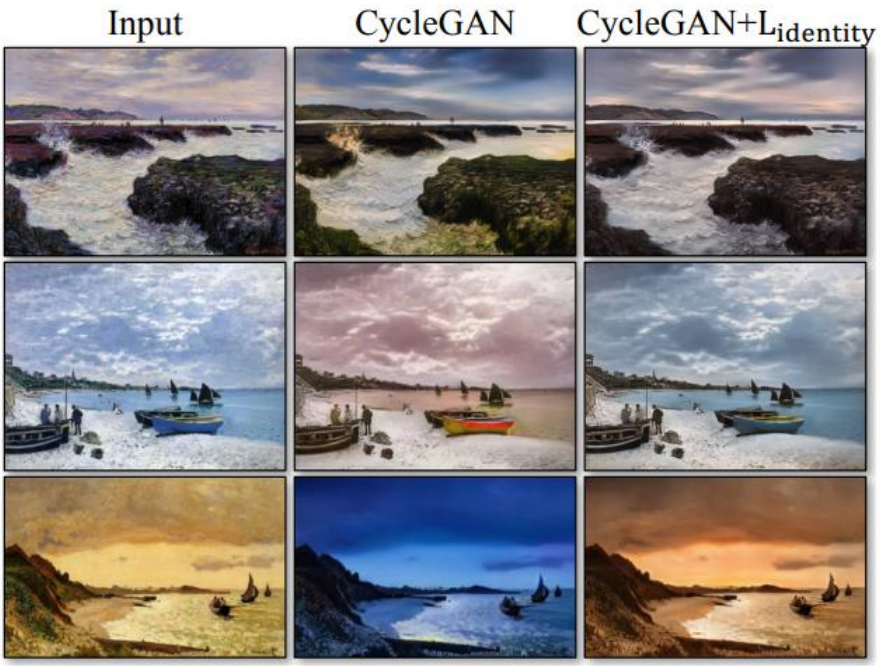

Figura 1.14: Esempio immagini generate con l'Identity Loss

CAPITOLO 2

Letteratura correlata

In questo secondo capitolo analizzeremo le metodologie principalmente utilizzate nel campo dell'Unpaired Image-to-Image Translation. In particolare, l'attenzione sarà focalizzata su architetture "Single-Modal output", dove un singolo input è "mappato" in modo deterministico su un unico output.

Dopo aver introdotto i concetti fondamentali relativi ai generatori e ai discriminatori tipicamente impiegati in questi approcci, passeremo ad esaminare dei lavori che utilizzano meccanismi basati sull'attenzione per migliorare i risultati dello stato dell'arte. In particolare, esamineremo due architetture distinte per dimostrare come i meccanismi di attenzione possano essere applicati in maniere diverse e non siano esclusivamente vincolati all'utilizzo dell'architettura specifica del Vision Transformer.

2.1 Metodologie in Unsupervised Image-to-Image Translation

Nel campo dell'Unsupervised Image-to-Image Translation, esistono vari modelli che generano un unico output in modo deterministico a partire da un input specifico. Questi modelli sono noti come *Single-modal Output*. In questa sezione, esploreremo le reti generative e avversarie più rilevanti in questo ambito.

2.1.1 La rete generatrice

L'architettura U-Net

L'architettura U-Net (Figura 2.1) è stata originariamente introdotta per problemi di segmentazione delle immagini [Ronneberger *et al.*, 2015]. È caratterizzata da una struttura encoder-decoder; la fase di encoding riduce le dimensioni spaziali delle immagini mantenendo le caratteristiche rilevanti, mentre la fase di decoding le rigenera. Una caratteristica distintiva della U-Net è la presenza di long-skip connection che collegano layer simili dell'encoder e del decoder. Questi collegamenti aiutano a mantenere i dettagli spaziali durante la fase di decodifica, migliorando, così, la qualità dell'immagine generata.

U-net nell'Unpaired Image-to-Image

CycleGAN [Zhu *et al.*, 2017] e Pix2Pix [Isola *et al.*, 2017] sono esempi di approcci che usano generatori basati sulla U-Net. In particolare, CycleGAN, che riprende l'idea da [Johnson *et al.*, 2016], utilizza un modello simile a U-Net per il suo Generatore, con l'aggiunta di blocchi ResNet come bottleneck. Per immagini di dimensioni 128×128 pixel, sono utilizzati 6 blocchi ResNet, mentre per immagini di dimensioni 256×256 pixel o superiori, il numero di blocchi

ResNet è aumentato a 9 (Figura 2.2). Anche il già citato articolo [Tang *et al.*, 2019b] utilizza un'architettura basata su una fase di encoder-decoder. Articoli come la GCGAN di Fu *et al.* [2018] si basano sullo stesso tipo di Generatore della CycleGAN, ma usano loss diverse dalla cycle-consistency per migliorarne i risultati.

Importanza della "Bottleneck"

Dopo la fase finale di encoding, i blocchi convoluzionali, o di altro tipo, costituiscono la "bottleneck" della rete generatrice; è questa una componente importante nell'architettura, che può essere costituita da blocchi convoluzionali, piuttosto che da transformer.

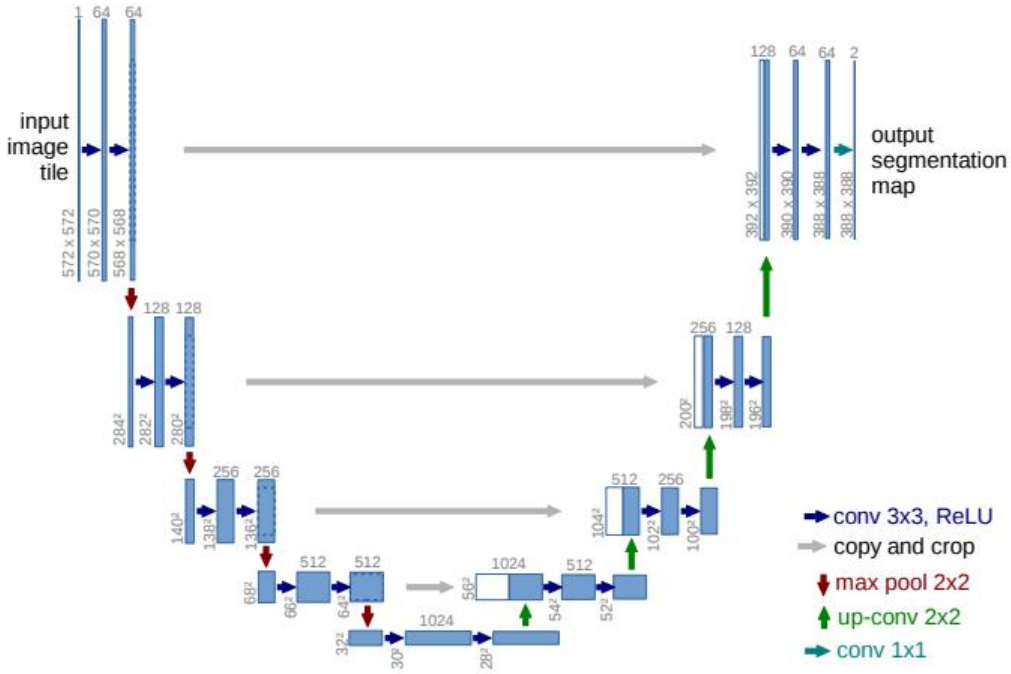

Figura 2.1: Architettura della U-Net

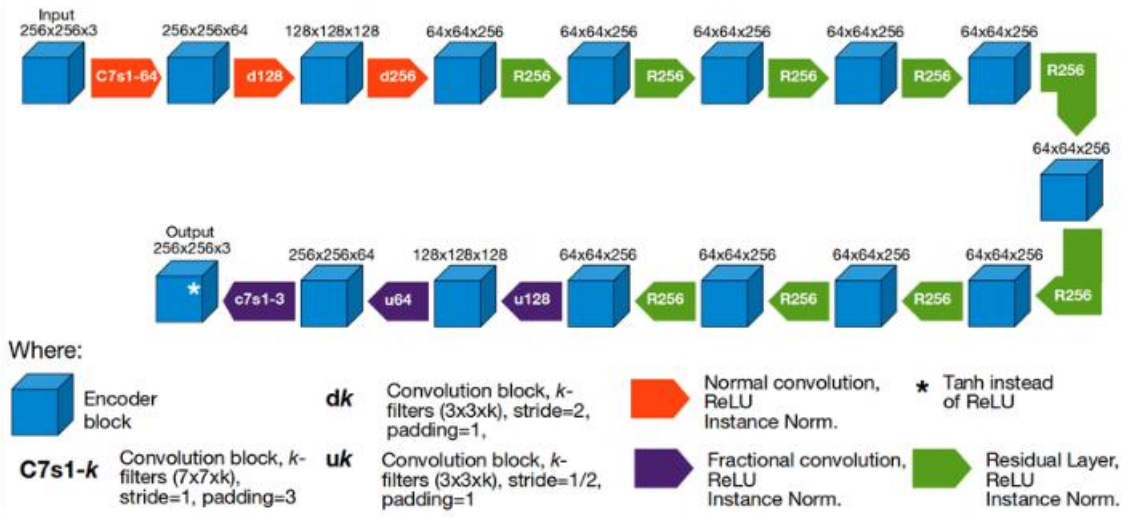

Figura 2.2: Architettura della CycleGAN

2.1.2 Il Discriminatore

Come già detto nel Capitolo 1, l'equilibrio tra Generatore e Discriminatore è di fondamentale importanza per ottenere buoni risultati.

Dal momento PatchGAN [Isola *et al.*, 2017] è molto usato nel dominio dell'Unpaired Image-to-Image, anche con architetture basate su transformer; pertanto, approfondiremo la sua architettura per capire le sue caratteristiche.

Architettura PatchGAN

Il Discriminatore PatchGAN (Figura 2.3) è composto da 5 layer convoluzionali. I layer iniziali utilizzano una convoluzione con stride 2 e padding 1, mentre gli ultimi due hanno stride 1 e padding 1. Questo design riduce gradualmente la dimensione spaziale dell'immagine, aumentando progressivamente le feature da 3 (RGB) a 512, attraverso la rete.

Dimensione dell'Output

Data un'immagine di input di dimensioni 256×256, PatchGAN produce un'immagine di output di dimensioni 30×30. Ogni valore in questa mappa 30×30 rappresenta un patch 70×70 dell'immagine di input. Matematicamente, se l'output è denotato come $O = D(x)$, dove D è il Discriminatore e x è l'immagine di input, allora $O(i, j)$ corrisponde al valore finale di un patch 70×70 dell'immagine originale.

Calcolo del Receptive Field

Il receptive field di un pixel è l'area dell'immagine di input che contribuisce al suo valore. Questa area può essere calcolata attraverso una serie di formule che considerano le dimensioni di input, il kernel, il padding e lo stride.

Siano:

- n_{in}: la dimensione dell'input;
- p: il padding;
- k: la dimensione del kernel;
- s: lo stride;
- j_{in}: il jump, ossia la distanza tra due feature adiacenti;
- r_{in}: il receptive field dell'input;

Le formule per calcolare la dimensione, il jump, il receptive field dell'output sono:

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1, \tag{2.1.1}$$

$$j_{out} = j_{in} \times s, \tag{2.1.2}$$

$$r_{out} = r_{in} + (k - 1) \times j_{in}. \tag{2.1.3}$$

Nel caso del Discriminatore PatchGAN, con 3 layer iniziali con $k = 4$, $s = 2$, $p = 1$, e gli ultimi due con $k = 4$, $s = 1$, $p = 1$, il receptive field può essere calcolato utilizzando le formule sopra indicate. Questo calcolo dimostra che ogni area del receptive field corrisponde a un patch 70×70 nell'immagine originale. Ciò comporta che ogni pixel legato al patch 70×70 dell'output $O(i, j)$ è indipendente da un altro pixel che dista 70 da esso.

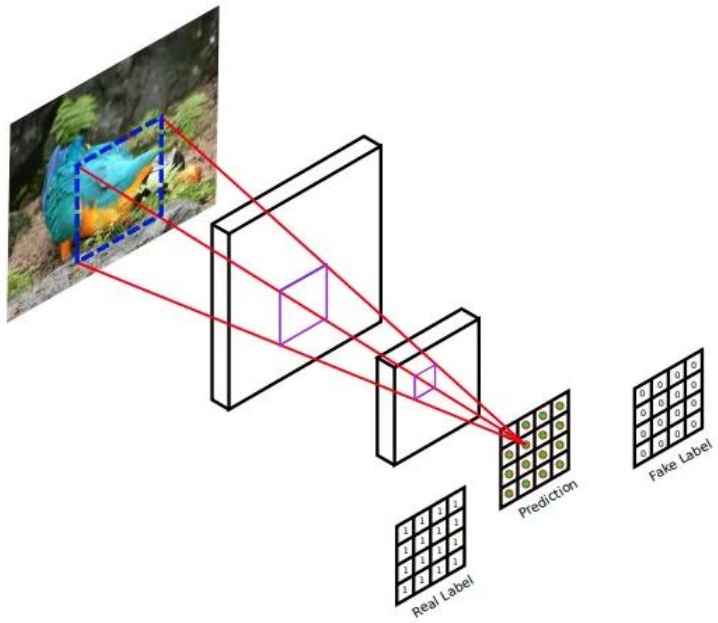

Figura 2.3: Illustrazione dell'architettura PatchGAN

2.2 Architetture con Meccanismi di Attenzione

Recentemente, nel campo dell'Unpaired Image-to-Image Translation, sono emersi nuovi modelli che integrano moduli di attenzione nei loro generatori e discriminatori. In questo contesto, focalizzeremo l'analisi su due architetture: U-GAT-IT [Kim *et al.*, 2019] e UVCGAN V1 [Torbunov *et al.*, 2022]. Entrambe queste reti utilizzano tecniche di attenzione per ottenere risultati notevoli.

Specificatamente, U-GAT-IT adotta un approccio di modulazione dell'attenzione che, pur differendo dai Transformer, mira a enfatizzare determinate feature nell'immagine durante il processo di traduzione tra domini diversi.

Invece, UVCGAN V1, usa un Generatore la cui bottlneck è basata fortemente sull'architettura del Vision Transformer.

Altre interessanti architetture basate sull'attenzione, come *AttentionGAN-v2* di [Tang *et al.*, 2021], sono elencate in [Pang *et al.*, 2021]

2.2.1 U-GAT-IT

U-GAT-IT è l'acronimo di Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. Di seguito approfondiremo gli aspetti chiave di questa architettura (Figura 2.4).

Strutture del Generatore e del Discriminatore

Il Generatore è composto da un encoder, un decoder e un classificatore ausiliario. L'encoder cattura le feature dell'immagine in input, che vengono poi trasformate dal decoder per generare un'immagine del dominio target. Il classificatore ausiliario, ispirato dalla Class Activation Mapping (CAM) di [Zhou *et al.*, 2016], è addestrato per concentrarsi su quelle feature, prodotte dall'encoder, che sono ancora vicine al dominio di partenza. La fase di decoding userà queste feature pesate per la traslazione al dominio finale.

Il Discriminatore, simile al Generatore (encoder-decoder), serve, invece, per distinguere tra immagini reali e generate. Anch'esso incorpora un classificatore ausiliario.

Meccanismo dell'Attenzione

Il classificatore ausiliario determina i pesi di attenzione per le mappe delle feature utilizzando il global average pooling e il global max pooling. L'output, forniti i layer, è il seguente:

$$\eta_s(x) = \sigma\left(\sum_k w_s^k \sum_{ij} E_s^{k_{ij}}(x)\right)$$

I pesi appresi dal classificatore vengono esplosi per creare l'input del decoder:

$$a_s(x) = \{w_s^k \times E_s^k(x) \mid 1 \leq k \leq n\}$$

Adaptive Layer-Instance Normalization (AdaLIN)

Il modello utilizza AdaLIN nei residual block del Generatore, che è una combinazione di Layer Normalization (LN), indicata nella formula da $\hat{a}_L$, e Instance Normalization (IN), che nella formula è $\hat{a}_I$:

$$\text{AdaLIN}(a, \gamma, \beta) = \gamma \cdot (\rho \cdot \hat{a}_I + (1 - \rho) \cdot \hat{a}_L) + \beta$$

ρ è un parametro addestrabile (tra 0-1) che bilancia il contributo di IN e LN.

CAM Loss

Tra le varie funzioni di loss utilizzate, come la GAN Loss, la cyclegan-consistency e l'identity-loss, già descritte nel primo capitolo, la CAM loss sfrutta i classificatori ausiliari sia nel Generatore che nel Discriminatore. Questa funzione di loss guida il modello nell'identificare regioni o feature nell'immagine che fanno la differenza più significativa tra i due domini nello stato attuale. Lo scopo è evidenziare quelle feature che sono più caratteristiche del dominio di origine, permettendo al modello di effettuare trasformazioni più efficaci nel dominio target.
Per il Generatore la CAM loss è data da:

$$\min_{\eta_s(x)} L_{\text{cam}}^{s \to t} = -\left(\mathbb{E}_{x \sim X_s}[\log(\eta_s(x))] + \mathbb{E}_{x \sim X_t}[\log(1 - \eta_s(x))]\right)$$

Per il Discriminatore, è la seguente:

$$\max_{\eta_D(x)} L_{\text{cam}}^{D} = \mathbb{E}_{x \sim X_l}[(\eta_D(x))^2] + \mathbb{E}_{x \sim X_s}[(1 - \eta_D(G_{s \to t}(x)))^2]$$

Queste loss indirizzano il modello a concentrarsi sulle feature più distintive tra i domini di origine e target, portando a traduzioni delle immagini più efficienti.

2.2.2 UVCGAN

La UVCGAN V1, acronimo di UNet-ViT GAN, trae ispirazione dalla CycleGAN, utilizzando le metodologie presentate nell'articolo [Zhu *et al.*, 2017]. Il Generatore è ulteriormente potenziato grazie all'integrazione di una bottleneck basata sull'architettura Vision Transformer, descritta in [Dosovitskiy *et al.*, 2020] (Figura 2.6).

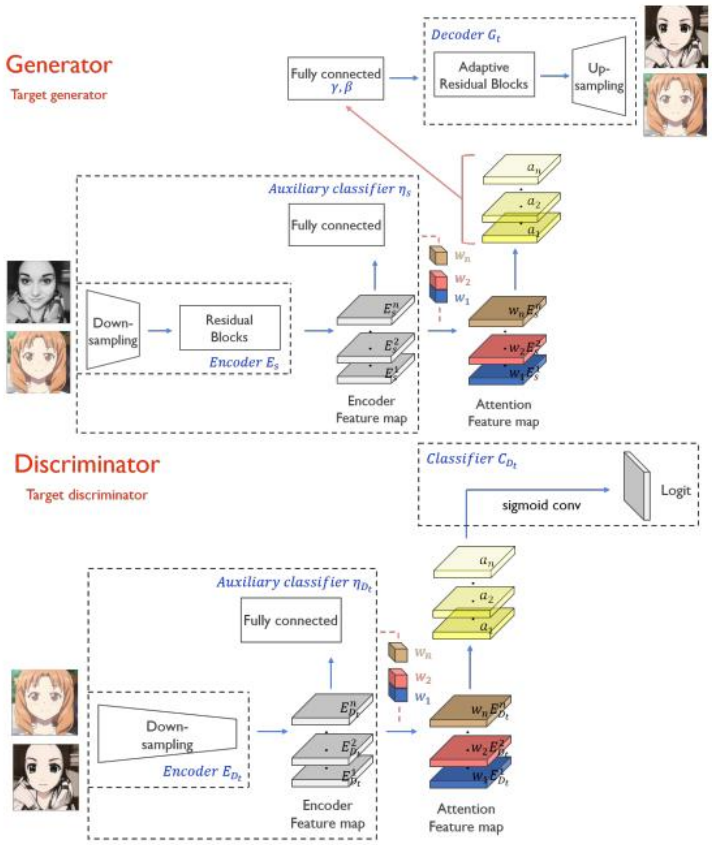

Figura 2.4: Illustrazione dell'Architettura U-GAT-IT

Generatore e Discriminatore

Il Generatore, come illustrato in Figura 2.6, segue una variante della U-Net. Invece di iniziare con 64 feature, questa versione inizia con 48 e aumenta progressivamente la dimensione fino a 384.

Dopo l'ultimo layer convoluzionale dell'encoder, l'immagine viene suddivisa in patch che vengono poi elaborate da una variante del ViT. Una delle principali novità di questa architettura consiste nell'uso di quello che viene definito come "Fourier Positional Embedding", introdotto nell'articolo [Tang *et al.*, 2019a]. In particolare, le coordinate spaziali normalizzate $x_{\text{norm}} = 2 \times \frac{x}{\text{width}-1} - 1$ e $y_{\text{norm}} = 2 \times \frac{y}{\text{height}-1} - 1$ vengono proiettate in uno spazio ad alta dimensionalità, prima di essere elaborate dalla funzione $sin(x)$, e infine concatenate ai token iniziali per catturare le relazioni spaziali tra i patch.

Come nel ViT, la normalizzazione del layer avviene prima della fase di *Multi-head Attention* e del modulo *Feed Forward*, utilizzando l'attivazione GeLU. In aggiunta, come tecnica di regolarizzazione, è presente un parametro allenabile α, inizialmente posto a zero, che modula l'ampiezza dei vettori in uscita sia dalla fase di attenzione che dai layer fully connected successivi.

Il decoder utilizza un up-sampling seguito da una convoluzione 2D per ridurre di metà le feature, le quali vengono poi concatenate (tramite long skip connection) alle feature dell'encoder.

Il Discriminatore rimane invece la PatchGAN 70x70, come per CycleGAN.

Figura 2.5: Esempi di risultati restituiti da U-GAT-IT

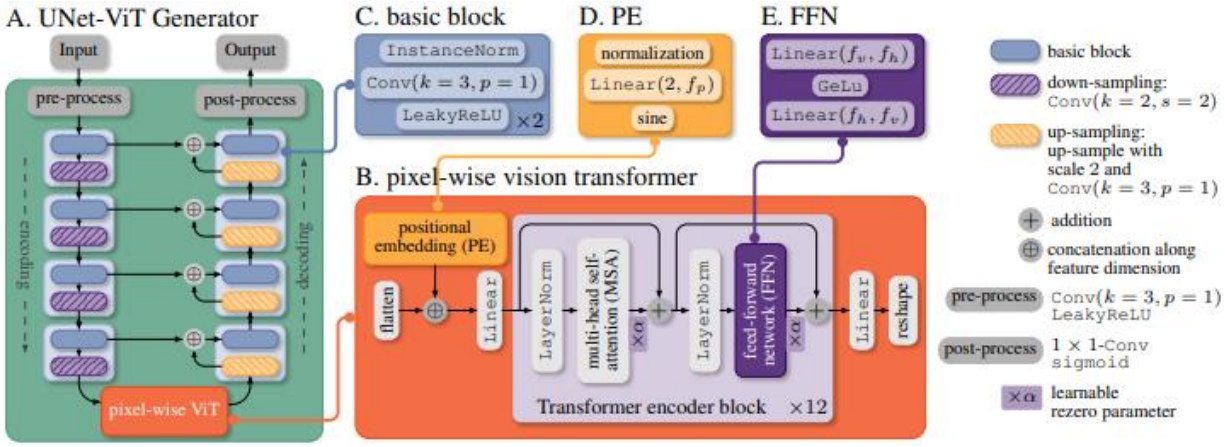

Figura 2.6: Architettura del Generatore UVCGAN v1

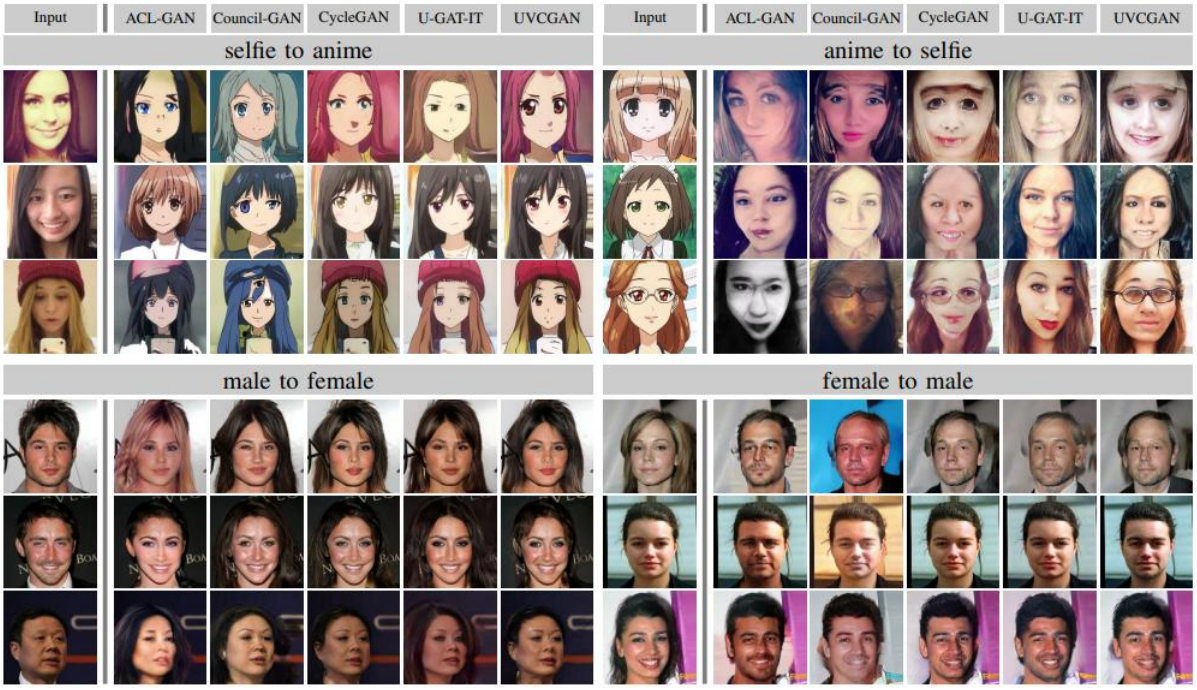

Figura 2.7: Esempi di risultati di UVCGAN v1

CAPITOLO 3

Definizione dell'Approccio

In questo capitolo verrà presentato e giustificato l'approccio scelto, ovvero l'adozione di una nuova rete avversaria denominata T-PatchGAN, che utilizza i meccanismi di attenzione del ViT per potenzialmente superare le prestazioni di PatchGAN, a parità di Generatore.

Seguirà una discussione dettagliata sulle tecniche di pre-processing dei dataset, sull'architettura complessiva della rete avversaria e sulle specifiche funzioni di loss utilizzate. Concluderemo con la descrizione della fase di training e delle metriche utilizzate per la valutazione delle prestazioni nella fase di testing.

3.1 Descrizione e Giustificazione dell'Obiettivo

La PatchGAN 70x70 è divenuta lo standard in diverse architetture popolari, inclusi CycleGAN, Pix2Pix, GCGAN, UVCGAN e altri.

L'obiettivo principale è di ideare una rete avversaria che incorpora i meccanismi di self-attention del Transformer, come delineato in [Vaswani *et al.*, 2017]. Questo modello avversario aspira a equilibrare efficienza computazionale e potenza espressiva, ponendosi come alternativa competitiva alla PatchGAN in diversi aspetti, quali parametri e prestazioni.

La complessità computazionale $O(n^2)$ dei meccanismi di attenzione nel Transformer rappresenta una sfida nota, specialmente quando si tratta di gestire un certo numero n di patch. Nonostante questo, l'obiettivo è di calibrare tale complessità in modo da non compromettere la qualità del modello, mentre si mira ad una efficienza simile a quella di PatchGAN.

Per quanto riguarda il Generatore, si ipotizza un equilibrio migliore se anch'esso basato su Transformer. Per tale ragione sarà adottata l'architettura di UVCGAN per i successivi esperimenti.

3.2 Pre-processing dei dati e Generatore

Utilizzando il Generatore delineato da UVCGAN v1 come riferimento, è essenziale sottolineare alcune pratiche di pre-processing dei dati descritte nel corrispondente paper. Questi passaggi sono importanti per migliorare la qualità dei risultati finali. Nonostante l'obiettivo principale sia confrontare la PatchGAN con la nuova architettura avversaria proposta, è fondamentale dettagliare questo aspetto.

3.2.1 Pre-processing e pre-training del Generatore

Se il dataset in uso non contiene immagini di dimensione 256 × 256, queste vengono ridimensionate a 256 × 256 pixel, prima del training o del testing, tramite interpolazione LANCZOS, una tecnica matematica per il ridimensionamento, utilizzato per minimizzare le distorsioni e gli artefatti. Le immagini sono inoltre normalizzate con valori compresi tra 0 e 1.

Per quanto riguarda l'inizializzazione dei pesi del Generatore, gli autori di UVCGAN v1 adottano un pre-training, che anche noi useremo. La figura, sempre dopo un ridimensionamento a 256 × 256 pixel, è divisa in patch non sovrapposti di dimensioni 32 × 32, e durante questa fase, il 40% della patch è mascherato. L'obiettivo è quello di ricostruire l'immagine originale.

L'ottimizzatore utilizzato in questo pre-training è Adam, in combinazione con uno scheduler basato su cosine annealing. Il cosine annealing modifica il learning rate dell'ottimizzatore durante la fase di training, facendolo variare in modo simile alla funzione coseno (Figura 3.1). Sono inoltre applicate diverse tecniche di data augmentation, quali rotazione casuale con piccoli angoli, cropping casuale, capovolgimento casuale e color jittering, ossia variazioni nei colori dell'immagine, dal contrasto alla saturazione.

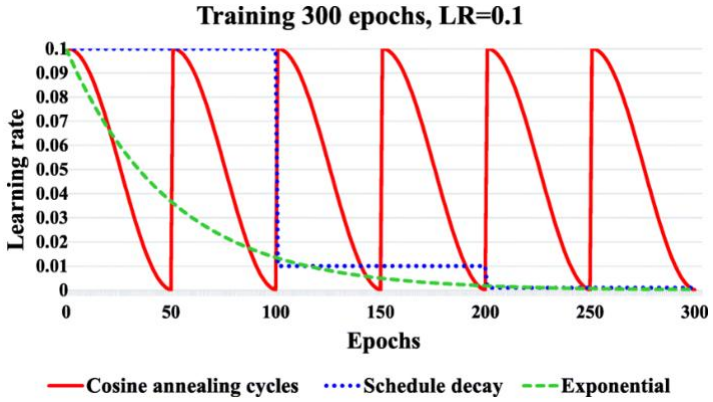

Figura 3.1: Illustrazione del funzionamento dello Scheduler Cosine Annealing

3.2.2 Generatore

Il Generatore, precedentemente discusso in generale nella Sezione 2.2.2, presenta ulteriori specificità che meritano approfondimento (Figura 3.2). L'immagine di partenza è prima elaborata da un layer iniziale che trasforma le feature a 48. Subito dopo inizia la fase dell'encoder, durante la quale ogni layer convoluzionale impiega due convoluzioni 2D e una convoluzione con stride $s = 2$, dimezzando di conseguenza le dimensioni spaziali dell'immagine. Le feature sono raddoppiate negli ultimi 3 blocchi dell'encoder. L'Instance Normalization (IN) è utilizzata come funzione di normalizzazione, sempre applicata dopo la funzione di attivazione LeakyRELU, sia nell'encoder che nel decoder.

L'Instance Normalization normalizza le feature map in modo indipendente per ogni esempio nel batch, a differenza della Batch Normalization che normalizza attraverso l'intero batch.L'Instance Normalization (IN) per una data feature map x può essere espressa attraverso l'equazione:

$$\mathrm{IN}(x) = \gamma \left(\frac{x - \mu(x)}{\sqrt{\sigma^2(x) + \epsilon}} \right) + \beta \tag{3.2.1}$$

dove $\mu(x)$ e $\sigma^2(x)$ sono, rispettivamente, la media e la varianza calcolate sull'intera feature map, ϵ è una costante piccola aggiunta per la stabilità numerica (per evitare la divisione per zero), e γ e β sono parametri appresi che rappresentano, rispettivamente, il coefficiente di scalatura e il bias di correzione. Dopo la fase di encoding, l'immagine, ridimensionata a $w_f = w_{\text{iniziale}}/16$ e $h_f = h_{\text{iniziale}}/16$, viene "appiattita" spazialmente. Il tensore di dimensione (B, C, H, W), dove B = batch_size, $C = 384$, $H = w_f$, $W = W_F$, diventa $(B, 384, w_f \times h_f)$. Queste dimensioni vengono, poi, permutate in $(B, w_f \times h_f, 384)$. A queste patch viene concatenato il Fourier Positional Encoder, risultando in uno spazio latente di dimensione $384 + f_p$, con $f_p = 384$. Un layer fully connected riporta la dimensione dello spazio di embedding a 384 (Figura 3.3).

Inoltre, oltre al pre-training del Generatore precedentemente descritto, gli autori [Torbunov *et al.*, 2022] utilizzano una Gradient Penalty (GP) definito come segue:

$$\mathcal{L}^{\text{GP}}_{\text{disc},A} = \mathcal{L}_{\text{disc},A} + \lambda_{\text{GP}} \mathbb{E}\left[\frac{\left(\|\nabla_x \mathcal{D}_A(x)\|_2 - \gamma\right)^2}{\gamma^2}\right],$$

dove $\mathcal{L}_{\text{disc},A}$ è la funzione di loss del Discriminatore per il dominio A. Questo termine di penalty del gradiente serve per migliorare la stabilità del training e renderlo meno sensibile alla scelta degli iperparametri.

Tuttavia, non utilizzaremo il GP per la PatchGAN e la nostra rete avversaria in quanto, secondo i risultati degli autori, non è essenziale per ottenere risultati di alta qualità se si esegue un adeguato pre-training del Generatore.

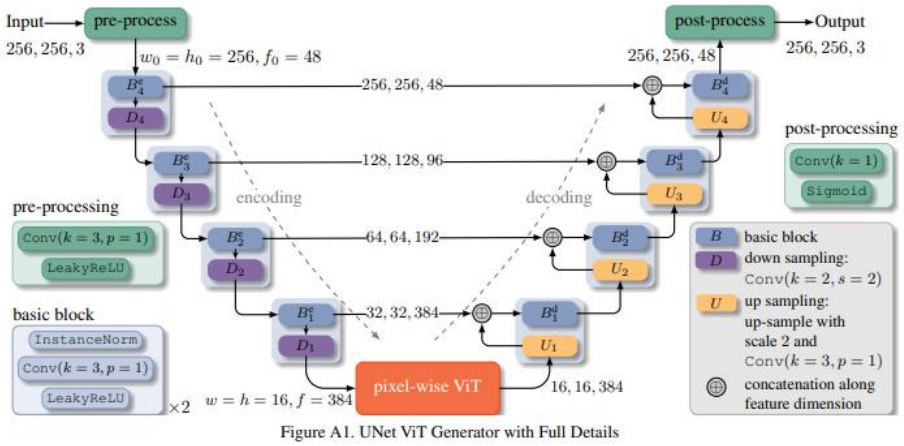

Figura 3.2: Encoder-Decoder UVCGAN v1

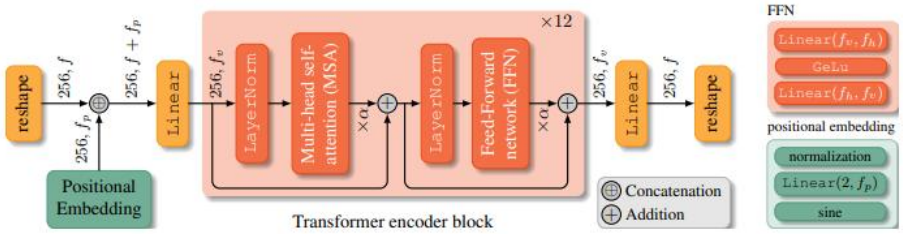

Figura 3.3: ViT UVCGAN v1

3.3 Rete Avversaria

L'obiettivo della rete avversaria è di categorizzare ogni patch dell'immagine come vero (1) o falso (0). A differenza del Vision Transformer, che analizza le relazioni globali tra tutti i patch, qui il focus sarà limitato all'analisi dei pixel all'interno di ogni singolo patch. Questa scelta è stata fatta per contenere i requisiti computazionali e per fornire una baseline che impedisca alla rete di apprendere troppo rapidamente rispetto al Generatore. Di conseguenza, l'output sarà composto da valori multipli, uno per ogni patch, piuttosto che da un singolo valore. A seguito della decisione di limitare l'analisi sui patch locali ai fini della classificazione, congiuntamente all'uso di un Transformer Encoder, abbiamo deciso di denominare l'architettura *T-PatchGAN*.

3.3.1 Architettura

L'architettura della rete avversaria è concepita come un ibrido tra layer convoluzionali e un Transformer Encoder. Di seguito, vengono dettagliate le varie componenti, mostrate in Figura 3.4.

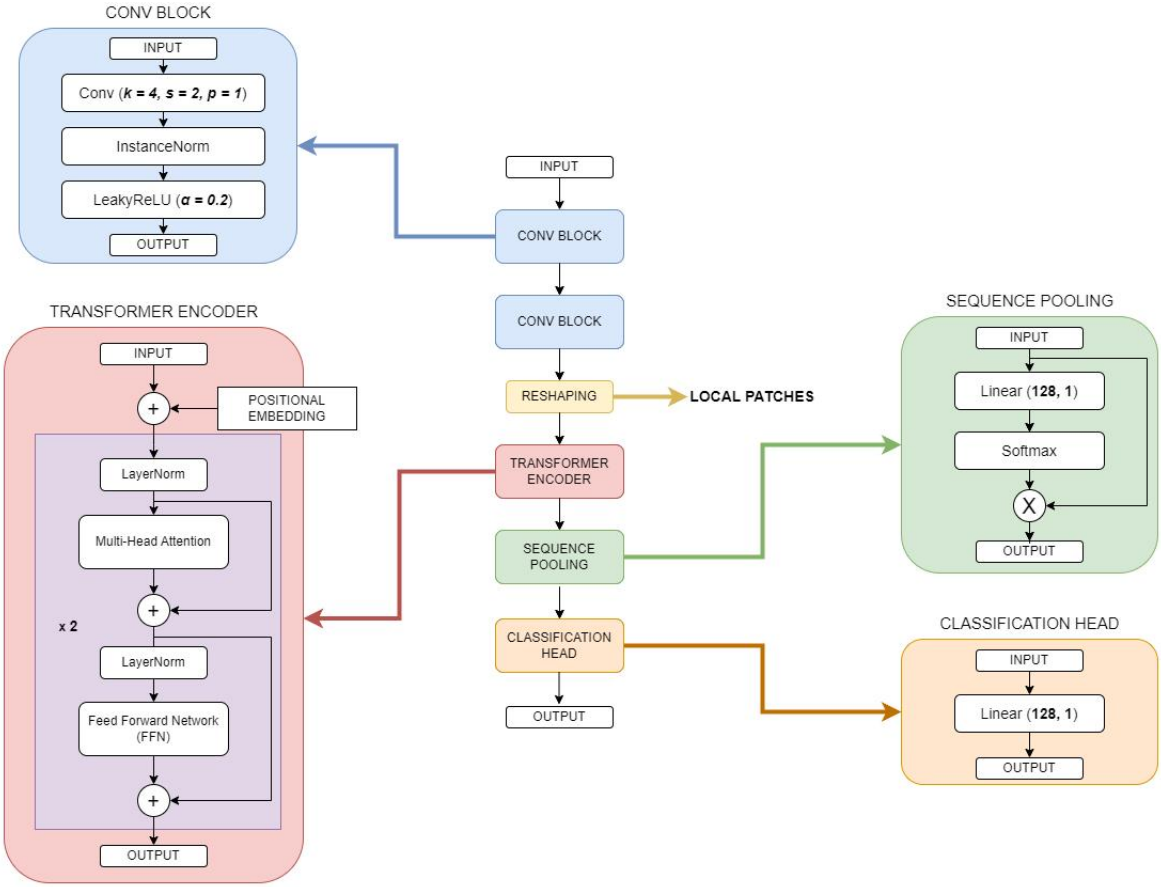

Figura 3.4: Architettura della T-PatchGAN

Layer Convoluzionali Iniziali

Il Discriminatore inizia con una serie di blocchi convoluzionali. Ogni blocco è strutturato nel seguente modo:

- *Conv2D*: dimensioni del kernel pari a 4×4, padding $p = 1$ e stride $s = 2$.
- *InstanceNormalization*: applicata per standardizzare le feature all'interno del batch su ogni canale separatamente.
- *LeakyReLU*: utilizzata come funzione di attivazione, con slope pari ad $\alpha = 0.2$.

Il numero di canali in uscita del primo blocco è 64, mentre il secondo blocco raddoppia questa dimensione, generando un tensore con 128 feature. Analogamente all'architettura della PatchGAN, la funzione di standardizzazione (*InstanceNormalization*) verrà applicata a partire dalla seconda convoluzione. Data un'immagine in ingresso di dimensioni 256×256, il tensore risultante sarà

$$(B, 128, 64, 64)$$

dove le dimensioni sono:

$$(B, C, H, W)$$

con $B =$ Batch Size, $C =$ Channels, $H =$ Height, $W =$ Width.

Transformer Encoder

Successivamente ai layer convoluzionali, il tensore viene processato per essere idoneo all'ingresso dei moduli del Transformer Encoder. Al fine di mantenere un equilibrio tra la potenza del Generatore e quella del Discriminatore, il tensore in uscita dai layer convoluzionali è segmentato in patch non sovrapposti. Ogni patch ha una dimensione patch_size^2, dove patch_size è un iperparametro inizialmente impostato a 4.

Partendo da un tensore di dimensioni $(B, 128, 64, 64)$, le patch vengono estratte e il tensore viene trasformato come

$$(B \times \text{numero\_patch}, \text{patch\_size}^2, 128)$$

dove

$$\text{numero\_patch} = \left(\frac{64}{\text{patch\_size}}\right)^2$$

Questo approccio può risultare computazionalmente oneroso per batch di grandi dimensioni. Tuttavia, poiché la dimensione del batch utilizzato sarà 1 (in accordo con l'architettura CycleGAN), l'impatto computazionale è minimo. Il Transformer adotta una struttura con 2 stacked layer e 2 head nel modulo *Multi-Head Attention*, e utilizza la funzione di attivazione GELU. Il dropout, applicato nel modulo *LayerNorm*, è disabilitato, avendo la rete pochi parametri, rispetto le classiche configurazioni del ViT. Infine, nel modulo *Feed Forward* moltplicheremo per 4 la dimensione dello spazio latente, per aumentare le capacità di apprendimento del modello.

Sequence Pooling e Classificazione

L'uscita del Transformer ha dimensione

$$(B \times \text{numero\_patch}, \text{patch\_size}^2, 128)$$

e *l'obiettivo finale* è classificare ciascun patch come 1 o 0, similmente al comportamento della PatchGAN 70x70, che genera un output finale di dimensioni 30×30 con un input di

dimensioni 256×256. Per fare ciò, useremo il meccanismo di *Sequence Pooling*, introdotto nel *Convolutional Compact Transformer* [Hassani *et al.*, 2021] con l'obiettivo di generare un solo output per patch, che tenga però conto di tutti i token (pixel del patch) in uscita dal Transformer. Per fare ciò, vengono calcolati pesi d'attenzione per ogni token (pixel), usando un layer fully connected, che prende l'output del Transformer:

$$\text{output} = (B \times \text{numero\_patch}, \text{patch\_size}^2, 128)$$

e lo converte in:

$$\text{attention} = (B \times \text{numero\_patch}, \text{patch\_size}^2)$$

Dopo aver applicato la funzione $softmax$ ad $attention$, viene calcolato il prodotto matriciale tra $attention$ e $output$, ottenendo il seguente tensore finale

$$\text{weighted\_output} = (B \times \text{numero\_patch}, 128)$$

dove lo spazio latente finale è, quindi, ottenuto sommando lo spazio latente pesato di tutti i pixel del patch. Utilizzando un altro layer *fully connected*, partendo da $weighted\_output$, si ottiene un tensore di dimensioni:

$$(B, \text{numero\_patch}, 1)$$

La terza dimensione rappresenta il valore che verrà classificato come 0 (fake) o 1 (real), a seconda se l'input al Discriminatore proviene, rispettivamente, dal Generatore o da una sorgente reale.

3.3.2 Numero di Parametri

La rete avversaria proposta, con un *patch_size* iniziale di 4, ha un numero di parametri allenabili relativamente basso, pari a 533.186, rispetto ai 2.766.529 della PatchGAN 70x70. Questa riduzione nel numero di parametri è contrastata dall'aumento della complessità computazionale introdotta dai due stacked encoder Transformer, nonché dalle 2 head nel modulo *Multi-Head Attention*.

3.3.3 Analogie con la PatchGAN

Similmente alla PatchGAN, la nostra rete avversaria produce un output con valori multipli, piuttosto che uno soltanto. Con un input di dimensione 256×256 e patch di dimensione 4×4 (area 16), si ottengono 256 valori finali, ciascuno dei quali classificato come 0 (fake) o 1 (real). In particolare, dal momento i layer convoluzionali della T-PatchGAN riducono di 4 le dimensioni originarie, un patch 4×4 corrisponde ad un'area di input 16×16.

Come nella PatchGAN quindi, il nostro modello è intrinsecamente focalizzato su patch locali. Questo contrasta la capacità di generalizzazione globale tipica dei modelli Vision Transformer (ViT). Tuttavia, è importante considerare il concetto di *Receptive Field*, discusso nella Sezione 2.1.2. Dopo i due blocchi convoluzionali iniziali, otteniamo un tensore di forma

$$(B, 128, 64, 64)$$

Applicando le Equazioni 2.1.2 e 2.1.3, il *Receptive Field* effettivo di ogni singolo pixel di un patch risulta essere di dimensione 10×10.

Per approfondire l'analisi dell'effetto della dimensione del patch sulle capacità di classificazione del modello, condurremo una serie di esperimenti utilizzando anche una dimensione di patch pari a 8×8, che aumenterà l'area dell'immagine di input analizzata per la classificazione a 32×32.

3.3.4 Inizializzazione dei pesi

PatchGAN

Per i layer convoluzionali di PatchGAN, seguendo le metodologie adottate da CycleGAN e UVCGAN v1, procediamo con l'inizializzazione dei pesi campionandoli da una distribuzione normale $\mathcal{N}(0, 0.02)$, con i bias impostati a zero. Invece, per i layer di normalizzazione come *InstanceNorm*, i coefficienti gamma (usati per "scalare" il valore normalizzato computato) sono inizializzati a uno e i bias sono mantenuti a zero. Tale approccio di inizializzazione mira a garantire che i layer normalizzati abbiano, in partenza, un effetto neutro sull'output del layer, facilitando la fase di apprendimento iniziale della rete.

T-PatchGAN

Nell'implementazione di T-PatchGAN, adotteremo un approccio coerente con quello utilizzato per PatchGAN per quanto riguarda l'inizializzazione dei layer di normalizzazione, come ad esempio *InstanceNorm*. Per gli altri layer, invece, sarà impiegata l'inizializzazione di tipo *Xavier* [Glorot e Bengio, 2010]. Questa strategia di inizializzazione è stata selezionata per la sua capacità di mantenere la varianza delle attivazioni costante durante le fasi di *forward* e di *backpropagation* della rete, contribuendo così alla stabilità dell'apprendimento. La formula per l'inizializzazione di Xavier è la seguente:

$$W \sim U\left(-gain\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, gain\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right) \tag{3.3.1}$$

dove W rappresenta i pesi da inizializzare, U indica una distribuzione uniforme, *gain* indica un fattore che dipende dalle funzioni di attivazione, n_j è il numero di unità nel layer j, e n_{j+1} è il numero di unità nel layer successivo.

Il fattore *gain* viene introdotto per ottimizzare l'inizializzazione in relazione alla funzione di attivazione utilizzata. Per esempio, per le funzioni di attivazione ReLU, il valore di *gain* raccomandato è $\sqrt{2}$.

3.4 Loss e Ottimizzatori utilizzati

Riprendendo le equazioni descritte nella Sezione 1.3.3, e aggiungendo l'Identity Loss, si ottiene:

$$\mathcal{L}(G, F, D_x, D_y) = \mathcal{L}_{\text{GAN}}(G, D_y, x, y) + \mathcal{L}_{\text{GAN}}(F, D_x, y, x) + \lambda\mathcal{L}_{\text{cyc}}(G, F) + \lambda_{\text{idt}}\mathcal{L}_{\text{idt}}(G, F) \tag{3.4.1}$$

$\mathcal{L}_{\text{GAN}}$, $\mathcal{L}_{\text{cyc}}$, e $\mathcal{L}_{\text{idt}}$ rappresentano, rispettivamente, la GAN Loss, la Cycle Consistency Loss, e l'Identity Loss.

Come nel contesto di UVCGAN e CycleGAN, per PatchGAN e T-PatchGAN adottiamo $\mathcal{L}_{idt} = \mathcal{L}_{\text{cyc}} =$ L1 Loss, mentre $\mathcal{L}_{\text{GAN}}$ = LSGAN. La LSGAN, introdotta in [Mao *et al.*, 2016], utilizza la L2 Loss al posto della Binary Cross Entropy (BCE) Loss. Questa scelta elimina anche la necessità di normalizzare l'output tramite una funzione sigmoide, consentendo valori nell'intervallo reale invece che limitati a [0,1].

Gli autori di UVCGAN suggeriscono di inizializzare λ_{cyc} tra 5 e 10, e λ_{idt}, se applicato, pari a metà di λ_{cyc}.

Infine, come ottimizzatore useremo Adam sia per i generatori che per i discriminatori, con un learning rate pari a 0.0001, che verrà mantenuto costante durante il training.

3.5 Definizione del training e testing

3.5.1 Data Augmentation

Nella fase di training, seguendo il procedimento adottato dagli autori di UVCGAN, si effettua un ridimensionamento casuale dell'immagine di un fattore 1.12. Questo significa che, partendo da un'immagine di dimensioni 256×256, si giunge a una dimensione di circa 286×286. Successivamente, si applica un *random cropping*, che seleziona e ingrandisce casualmente alcune regioni dell'immagine. Infine, si esegue un capovolgimento orizzontale casuale dell'immagine.

3.5.2 Batch Size e Iterazioni

Il batch size viene mantenuto a 1, in accordo con le specifiche di CycleGAN. Durante ogni iterazione di training, prima si aggiornano i discriminatori e poi i generatori.

3.5.3 Buffer Immagini per i Discriminatori

Al fine di ottimizzare la stabilità del training, adotteremo la metodologia proposta da [Shrivastava *et al.*, 2016], impiegando un buffer di x immagini per ciascun Discriminatore. In linea con CycleGAN e UVCGAN v1, porremo $x = 50$. Il buffer mantiene in memoria le immagini generate negli ultimi 50 step dal Generatore. Ad ogni iterazione, c'è una probabilità del 50% che un'immagine casuale dal buffer venga scelta per l'aggiornamento del Discriminatore, venendo poi sostituita dall'immagine corrente prodotta dal Generatore. Altrimenti, il Discriminatore vedrà l'output attuale del Generatore.

3.5.4 Aggiornamento dei Pesi

Una volta selezionate le immagine dai buffer dei due discriminatori, i pesi della rete avversaria vengono aggiornati per minimizzare la GAN Loss. Successivamente, il Generatore cerca di massimizzare questa loss, nel tentativo di "ingannare" il Discriminatore. In aggiunta, vengono applicate la Cycle-Consistency Loss e, ove necessario, l'Identity Loss per aggiornare i pesi dei generatori.

3.5.5 Composizione delle Loss

La funzione di loss totale utilizzata per l'aggiornamento dei pesi è una combinazione di GAN Loss, Cycle-Consistency Loss e Identity Loss, come descritto nell'Equazione 3.4.1.

3.5.6 Metriche per il Testing

Per valutare la qualità delle immagini generate durante il testing, utilizziamo la Kernel Inception Distance (KID) [Bińkowski *et al.*, 2018]. La KID utilizza il modello Inception V3 per estrarre le caratteristiche dalle immagini. In particolare, le feature vengono calcolate utilizzando l'ultimo strato convoluzionale di Inception V3, che è pre-allenato su ImageNet. La KID è data dalla seguente equazione:

$$\mathrm{KID} = \mathbb{E}[k(x, x')] + \mathbb{E}[k(y, y')] - 2\mathbb{E}[k(x, y)] \tag{3.5.1}$$

dove x e x' sono vettori di feature indipendenti estratti dalle immagini reali, y e y' sono vettori di feature indipendenti estratti dalle immagini generate, e $k(\cdot, \cdot)$ è la funzione kernel polinomiale definita come segue:

$$k(x, y) = (x^T y + c)^{\text{degree}} \tag{3.5.2}$$

dove $x^T y$ è il prodotto tra i due vettori di feature, c è una costante che può essere aggiustata, e *degree* è il grado del kernel polinomiale. La scelta del kernel e dei suoi parametri (c e *degree*) può influenzare la sensibilità della KID rispetto alle diverse caratteristiche delle distribuzioni delle immagini. Questa misurazione offre una stima della divergenza tra le distribuzioni delle immagini reali e quelle generate.

È importante notare che Inception V3, pre-allenata su ImageNet, è ottimizzata per immagini di dimensioni 299×299. Nel nostro caso, le immagini utilizzate per il training, dopo eventuale reshaping tramite LANCZOS, hanno dimensioni 256×256; di conseguenza, anche quelle generate hanno la stessa dimensione finale. Per ovviare a ciò, verrà applicato un resize alla dimensione richiesta dal modello Inception V3 dalla libreria `https://github.com/toshas/torch-fidelity`, che useremo per il calcolo della metrica.

CAPITOLO 4

Implementazione in Python della Rete Avversaria

Questo capitolo descrive l'ambiente di lavoro utilizzato e l'implementazione in PyTorch della rete avversaria proposta. Inoltre, dettaglieremo le librerie utilizzate per il pre-processing dei dati, la configurazione del generatore UNet-ViT, e il calcolo della metrica KID.

4.1 Ambiente di lavoro

Abbiamo utilizzato Jupyter Notebook come ambiente di sviluppo, avvalendoci delle GPU fornite da Google Colab e Kaggle. Il codice è stato scritto in Python 3.10.12. Per la manipolazione delle immagini, la definizione dei modelli e il loro addestramento, abbiamo usato le librerie `torch`, `torchvision` e `PIL`. Le versioni specifiche di torch sono 2.1.0+cu118 su Google Colab e 2.0.0 su Kaggle.

4.2 Dataset

Il dataset necessario per il pre-training e il pre-processing, come descritto nella Sezione 3.2.1, è stato inizializzato estendendo la classe `torch.utils.data.Data`. Per la data augmentation, abbiamo utilizzato i metodi di `torchivision.transforms`. Per il pre-training, la composizione delle trasformazioni è la seguente:

```
self.transforms = transforms.Compose([
            transforms.Resize(int(self.img_size[0] * 1.12),
             ↪  Image.LANCZOS),
            transforms.RandomCrop(self.img_size[0]),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(10),
            transforms.ColorJitter(brightness=0.2, contrast=0.2,
             ↪  saturation=0.2, hue=0.2)
        ])
```

Per il pre-processing dell'immagine, ai fini del training, abbiamo utilizzato i seguenti metodi:

```
 self.transforms = transforms.Compose([
          transforms.Resize(int(self.img_size * 1.12),
           ↪  Image.LANCZOS),
```

```
            transforms.RandomCrop(self.img_size),
            transforms.RandomHorizontalFlip(),
        ])
```

Durante la fase di test, l'immagine non viene trasformata; a tal fine viene introdotta la variabile `self.test`, che sarà `True` in caso di testing, e `False` altrimenti:

```
if not self.test:
    image = self.transforms(image)
```

Invece, per il resizing delle immagini abbiamo usato il metodo `LANCZOS` della libreria `PIL` come segue:

```
from PIL import Image

img = img.resize((w, h), Image.LANCZOS)
```

4.3 Generatore

Il generatore utilizzato deriva dall'articolo UVCGAN v1. Abbiamo importato la classe `ViTUNetGenerator` dal repository GitHub ufficiale `https://github.com/LS4GAN/uvcgan`, specificamente da `uvcgan.models.generator.vitunet`:

```
from uvcgan.models.generator.vitunet import ViTUNetGenerator

# Device setting
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Initialize the Generators

generator_A_to_B = ViTUNetGenerator(
    features=384,
    n_heads=6,
    n_blocks=6,
    ffn_features=1536,
    embed_features=384,
    activ='gelu',
    norm='layer',
    image_shape=(3, 256, 256),
    unet_features_list=[48, 96, 192, 384],
    unet_activ='leakyrelu',
    unet_norm='instance',
    unet_downsample='conv',
    unet_upsample='upsample-conv',
    unet_rezero=False,
    rezero=True,
    activ_output='sigmoid'
).to(device)

generator_B_to_A = ViTUNetGenerator(
    features=384,
```

```
    n_heads=6,
    n_blocks=6,
    ffn_features=1536,
    embed_features=384,
    activ='gelu',
    norm='layer',
    image_shape=(3, 256, 256),
    unet_features_list=[48, 96, 192, 384],
    unet_activ='leakyrelu',
    unet_norm='instance',
    unet_downsample='conv',
    unet_upsample='upsample-conv',
    unet_rezero=False,
    rezero=True,
    activ_output='sigmoid'
).to(device)
```

4.4 T-PatchGAN

Per ciò che riguarda la rete avversaria, abbiamo creato una classe `Discriminator`, la quale estende `torch.nn.Module`. Nell'istanziazione della sottoclasse abbiamo definito i vari parametri allenabili, che verranno, poi, usati in fase di training e validazione, tramite il metodo `forward`, per produrre l'immagine traslata da un dominio all'altro.

Di seguito l'implementazione dell'intera classe:

```
class Discriminator(nn.Module):
    def __init__(self, img_size: int = 256, patch_size: int = 4,
     ↪  channels: int = 3, transformer_layers: int = 2, heads: int
     ↪  = 4):
        """
        Discriminator constructor.

        Args:
        img_size (int, optional): The size of the input image.
         ↪  Default is 256.
        patch_size (int, optional): The size of each image patch
         ↪  for the Transformer. Default is 4.
        channels (int, optional): The number of channels in the
         ↪  input image. Default is 3.
        transformer_layers (int, optional): The number of
         ↪  Transformer encoder layers. Default is 2.
        heads (int, optional): The number of attention heads in the
         ↪  Transformer encoder. Default is 4.
        """
        super().__init__()

        self.patch_size = patch_size
        self.num_patches = (img_size // 4 // patch_size) ** 2
        self.embed_dim = 128
```

```
        # Initial Convolutional Layers
        self.conv_layers = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=4, stride=2,
             ↪  padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2)
        )

        # Transformer Configuration
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=self.embed_dim,
            nhead=heads,
            dim_feedforward=self.embed_dim * 4,
            dropout=0.0,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )
        self.transformer_encoder =
         ↪  nn.TransformerEncoder(encoder_layer,
         ↪  num_layers=transformer_layers)

        # Trainable Positional Embedding
        self.positional_embedding = nn.Parameter(torch.zeros(1,
         ↪  self.patch_size ** 2, self.embed_dim))

        # Attention Pooling Layer
        self.attention_pool = nn.Linear(self.embed_dim, 1)

        # Fully Connected Layer
        self.fc = nn.Linear(self.embed_dim, 1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Forward pass through the network.

        Args:
        x (torch.Tensor): Input tensor of shape [batch_size,
         ↪  channels, height, width].

        Returns:
        torch.Tensor: Output tensor.
        """
        # Pass through Conv layers
        x = self.conv_layers(x)

        # Shape variables
```

```
        b, c, h, w = x.size()

        # Create non-overlapping patches
        x = x.unfold(2, self.patch_size, self.patch_size).unfold(3,
         ↪  self.patch_size, self.patch_size)

        # Reorder dimensions and reshape to get local patches as
         ↪  tokens
        x = x.permute(0, 2, 3, 4, 5, 1).contiguous().view(-1,
         ↪  self.patch_size ** 2, self.embed_dim)

        # Add positional encoding to pixels of the patch
        x = x + self.positional_embedding

        # Pass through Transformer encoder
        x = self.transformer_encoder(x)  # Output shape:
         ↪  [batch_size * num_patches, patch_size ** 2, embed_dim]

        # Sequence Pooling
        attn_weights = self.attention_pool(x).squeeze(-1)  # Output
         ↪  shape: [batch_size * num_patches, patch_size ** 2]
        x = torch.einsum('b n, b n d -> b d',
         ↪  attn_weights.softmax(dim=1), x)  # Output shape:
         ↪  [batch_size * num_patches, embed_dim]

        # Reshape for final layer
        x = x.view(b, self.num_patches, -1)  # Output shape:
         ↪  [batch_size, num_patches, embed_dim]

        # Pass through Fully Connected layer
        x = self.fc(x)  # Output shape: [batch_size, num_patches,
         ↪  1]

        return x
```

Come si può notare in `__init__`, viene calcolato in anticipo il numero di patch (`num_patches`) che diventeranno poi i vari input in ingresso al Transformer Encoder per ogni immagine del batch, in questo caso una soltanto. Attraverso `torch.nn.Sequential` abbiamo unito i due blocchi convoluzionali i quali, partendo da un input di dimensioni 256×256, e con i 3 canali RGB in ingresso, produrranno il tensore con shape:

dove le dimensioni sono:

$$(\text{Batch Size}, \text{Channels}, \text{Height}, \text{Width})$$

Utilizzando l'output dei layer convoluzionali, procediamo al reshaping del tensore per fornire in input al Transformer i patch locali. I token sono rappresentati dai singoli pixel all'interno dei patch e il reshaping viene effettuato utilizzando il metodo `torch.tensor.unfold`, definito come segue:

$$Tensor.unfold(dimension, size, step) \rightarrow Tensor$$

Impostiamo $dimension = 2$, $size =$ patch_size e $step =$ patch_size, e partiamo dalla seconda dimensione, ovvero `Height`. In questo modo, dividiamo la feature in patch di dimensioni `patch_size`, mantenendo ogni patch isolata dalle altre.

Per illustrare meglio il processo, consideriamo una matrice di dimensioni $(1, 3, 4, 4)$ e usiamo patch_size $= 2$:

```
init_t = torch.randn((1,3,4,4))
print("Initial tensor:","\n", init_t)
dimension = 2
patch_size = 2
step = 2
t=init_t.unfold(dimension, patch_size, step)
print("\n Output shape:", t.shape)
print("\n Output tensor:","\n", t)
```

I risultati sono i seguenti:

```
Initial tensor:
 tensor([[[[ 0.0054, -0.3822,  1.2420, -0.9512],
           [-0.1694, -0.9628,  0.7597, -0.7096],
           [-0.6743, -1.1654,  2.2036, -0.3548],
           [ 0.2094,  0.2265, -0.7965,  2.7719]],

          [[ 0.3947, -0.4681, -1.4231, -0.0505],
           [ 0.1921, -1.0474,  0.7394, -0.3575],
           [-1.5611,  1.2005, -0.7313,  0.3147],
           [-0.0969, -0.4339,  0.9257,  0.0979]],

          [[ 0.5820, -0.4566, -1.8331, -1.6134],
           [ 0.9316,  0.8116,  0.4069, -0.5991],
           [-0.5961, -1.4099, -2.1124, -0.7662],
           [-0.0945, -1.0799,  2.0511, -0.3313]]]])

 Output shape: torch.Size([1, 3, 2, 4, 2])

 Output tensor:
 tensor([[[[[ 0.0054, -0.1694],
            [-0.3822, -0.9628],
            [ 1.2420,  0.7597],
            [-0.9512, -0.7096]],

           [[-0.6743,  0.2094],
            [-1.1654,  0.2265],
            [ 2.2036, -0.7965],
            [-0.3548,  2.7719]]],


          [[[ 0.3947,  0.1921],
```

```
           [-0.4681, -1.0474],
           [-1.4231,  0.7394],
           [-0.0505, -0.3575]],

          [[-1.5611, -0.0969],
           [ 1.2005, -0.4339],
           [-0.7313,  0.9257],
           [ 0.3147,  0.0979]]],


         [[[ 0.5820,  0.9316],
           [-0.4566,  0.8116],
           [-1.8331,  0.4069],
           [-1.6134, -0.5991]],

          [[-0.5961, -0.0945],
           [-1.4099, -1.0799],
           [-2.1124,  2.0511],
           [-0.7662, -0.3313]]]])
```

Dal momento che `Height` = 4, il tensore viene scomposto risultando in un'ulteriore dimensione pari a $4/2 = 2$. Applicando lo stesso procedimento lungo la dimensione $= 3$, cioè `Width`, otteniamo:

```
Initial tensor:
 tensor([[[[ 2.5842,  1.1971,  0.7919, -0.3393],
           [ 0.4368,  0.2745, -0.2163,  2.1870],
           [-1.7612, -0.3867, -1.6483, -0.6802],
           [-1.1203, -0.8100, -2.1600,  1.6300]],

          [[-0.1306, -1.2264,  0.4104,  0.2961],
           [-0.5618,  1.0354, -0.5322, -0.4036],
           [ 0.8276, -0.2494, -1.4766, -0.0871],
           [ 1.0169,  0.3055,  1.0815,  0.5113]],

          [[-0.6112, -0.6697,  1.0302,  0.7782],
           [ 0.5746,  0.5049,  0.5953,  0.1125],
           [ 0.6725, -0.8818,  1.0289, -0.4599],
           [ 1.1600, -0.4128, -1.2385, -0.5141]]]])

 Output shape: torch.Size([1, 3, 2, 2, 2, 2])

 Output tensor:
 tensor([[[[[[ 2.5842,  1.1971],
             [ 0.4368,  0.2745]],

            [[ 0.7919, -0.3393],
             [-0.2163,  2.1870]]],
```

```
          [[[-1.7612, -0.3867],
            [-1.1203, -0.8100]],

           [[-1.6483, -0.6802],
            [-2.1600,  1.6300]]]],



        [[[[-0.1306, -1.2264],
           [-0.5618,  1.0354]],

          [[ 0.4104,  0.2961],
           [-0.5322, -0.4036]]],


         [[[ 0.8276, -0.2494],
           [ 1.0169,  0.3055]],

          [[-1.4766, -0.0871],
           [ 1.0815,  0.5113]]]],



        [[[[-0.6112, -0.6697],
           [ 0.5746,  0.5049]],

          [[ 1.0302,  0.7782],
           [ 0.5953,  0.1125]]],


         [[[ 0.6725, -0.8818],
           [ 1.1600, -0.4128]],

          [[ 1.0289, -0.4599],
           [-1.2385, -0.5141]]]]])
```

Il risultato è quello atteso: le varie feature (3 in questo caso) sono divise in 4 patch (dato che $patch\_size = (4/2)^2$), ciascuna con $H = 2$ righe e $W = 2$ colonne. Di conseguenza, abbiamo un tensore con dimensioni:

$$(1, 3, 2, 2, 2, 2)$$

Infine, per ottenere il tensore con i singoli pixel (token), dove ogni token ha le sue 128 feature che costituiscono lo spazio di embedding, utilizziamo i metodi `permute` e `view` nel seguente modo:

```
t = t.permute(0,2,3,4,5,1).contiguous().view(-1, patch_size**2, 3)
print("Initial tensor:","\n", init_t)
print("\n Output shape:", t.shape)
print("\n Output tensor:","\n", t)
```

I risultati sono i seguenti:

```
    Initial tensor:
 tensor([[[[-1.5751e+00, -4.1372e-01, -1.2695e-03,  4.2855e-01],
           [-1.4207e+00,  9.2825e-01, -1.9404e-01, -4.9099e-01],
           [-7.1796e-01,  8.7322e-01, -1.1545e+00,  1.6569e+00],
           [-5.1420e-01,  1.7331e+00, -1.1534e+00,  2.1033e-01]],

          [[ 3.8797e-01, -1.7482e+00,  3.5356e-03,  9.1257e-01],
           [-1.1840e+00, -1.6451e+00,  1.4959e+00,  7.1703e-01],
           [ 2.6221e+00,  6.6826e-01,  8.0883e-01,  1.1272e+00],
           [ 4.5492e-01,  1.7154e+00,  1.4644e+00, -1.5816e-01]],

          [[-6.5528e-01,  8.3392e-01,  1.0368e+00,  1.1980e+00],
           [ 1.4908e+00,  2.1416e-01,  6.7554e-01, -1.0965e+00],
           [ 8.6397e-01, -1.1166e+00,  5.2159e-01, -6.3592e-01],
           [-1.0247e+00, -6.6544e-01,  1.3028e-01,  8.8464e-01]]]])

 Output shape: torch.Size([4, 4, 3])

 Output tensor:
 tensor([[[-1.5751e+00,  3.8797e-01, -6.5528e-01],
          [-4.1372e-01, -1.7482e+00,  8.3392e-01],
          [-1.4207e+00, -1.1840e+00,  1.4908e+00],
          [ 9.2825e-01, -1.6451e+00,  2.1416e-01]],

         [[-1.2695e-03,  3.5356e-03,  1.0368e+00],
          [ 4.2855e-01,  9.1257e-01,  1.1980e+00],
          [-1.9404e-01,  1.4959e+00,  6.7554e-01],
          [-4.9099e-01,  7.1703e-01, -1.0965e+00]],

         [[-7.1796e-01,  2.6221e+00,  8.6397e-01],
          [ 8.7322e-01,  6.6826e-01, -1.1166e+00],
          [-5.1420e-01,  4.5492e-01, -1.0247e+00],
          [ 1.7331e+00,  1.7154e+00, -6.6544e-01]],

         [[-1.1545e+00,  8.0883e-01,  5.2159e-01],
          [ 1.6569e+00,  1.1272e+00, -6.3592e-01],
          [-1.1534e+00,  1.4644e+00,  1.3028e-01],
          [ 2.1033e-01, -1.5816e-01,  8.8464e-01]]])
tensor([[[-1.5751e+00,  3.8797e-01, -6.5528e-01],
          [-4.1372e-01, -1.7482e+00,  8.3392e-01],
          [-1.4207e+00, -1.1840e+00,  1.4908e+00],
          [ 9.2825e-01, -1.6451e+00,  2.1416e-01]],

         [[-1.2695e-03,  3.5356e-03,  1.0368e+00],
          [ 4.2855e-01,  9.1257e-01,  1.1980e+00],
          [-1.9404e-01,  1.4959e+00,  6.7554e-01],
          [-4.9099e-01,  7.1703e-01, -1.0965e+00]],

         [[-7.1796e-01,  2.6221e+00,  8.6397e-01],
          [ 8.7322e-01,  6.6826e-01, -1.1166e+00],
```

```
          [-5.1420e-01,  4.5492e-01, -1.0247e+00],
          [ 1.7331e+00,  1.7154e+00, -6.6544e-01]],

         [[-1.1545e+00,  8.0883e-01,  5.2159e-01],
          [ 1.6569e+00,  1.1272e+00, -6.3592e-01],
          [-1.1534e+00,  1.4644e+00,  1.3028e-01],
          [ 2.1033e-01, -1.5816e-01,  8.8464e-01]]])
```

Come è possibile osservare, il tensore che sarà fornito in ingresso al Transformer Encoder sarà caratterizzato da una `lunghezza` corrispondente al numero di pixel contenuti in ciascun patch. I singoli pixel fungeranno da `token`. Nel caso specifico di questo esempio, abbiamo che `lunghezza` = 4 e `token` = 4. Le diverse feature associate a ciascun pixel costituiranno lo spazio di embedding (o spazio latente). Questo spazio sarà sfruttato dalle 2 head presenti nel modulo *Multi-Head Attention* per calcolare "attenzioni" differenti. Successivamente, queste attenzioni verranno concatenate al fine di ottenere un unico tensore in uscita con dimensioni:

$$(B \times \text{numero\_patch}, \text{patch\_size}^2, \text{Embedding})$$

Questo tensore verrà, poi, processato ulteriormente dal modulo *Feed-Forward*.

Prima di essere inserito nel Transformer Encoder, al tensore viene sommato un learnable *Positional Embedding*, implementato mediante `torch.nn.parameter.Parameter`: i valori dei parametri iniziali vengono impostati a zero, così da non influenzare le feature apprese dai layer convoluzionali. Durante le fasi di backpropagation, questi valori verranno automaticamente aggiustati allo scopo di minimizzare le funzioni di loss applicate.

Successivamente al passaggio attraverso il Transformer, il modulo *Sequence Pooling* viene utilizzato per calcolare l'attenzione associata a ciascun pixel di ogni patch. Tale calcolo viene effettuato mediante l'utilizzo di un layer lineare:

```
attn_weights = self.attention_pool(x).squeeze(-1) # Output shape:
 ↪  [batch_size * num_patches, patch_size ** 2]
```

A questo layer viene applicata la funzione `Softmax` prima del prodotto matriciale con l'output del Transformer Encoder:

```
x = torch.einsum('b n, b n d -> b d', attn_weights.softmax(dim=1),
 ↪  x) # Output shape: [batch_size * num_patches, embed_dim]
```

Facendo un reshaping del tensore attraverso il codice:

```
x = x.view(b, self.num_patches, -1)  # Output shape: [batch_size,
 ↪  num_patches, embed_dim]
```

e applicando un layer lineare sulla dimensione di embedding, otteniamo il tensore finale, ovvero

$$\text{output} = (\text{B}, \text{numero\_patch}, 1)$$

4.5 Inizializzazione dei Pesi

4.5.1 PatchGAN

Per la PatchGAN, il codice usato per l'inizializzazione dei pesi è il seguente:

```
def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find('Conv2d') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Norm') != -1:  # This will cover
     ↪  BatchNorm, InstanceNorm, and LayerNorm
        if hasattr(m, 'weight') and m.weight is not None:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Linear') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)


# Apply the weights initialization function to the discriminator
discriminator_A.apply(weights_init_normal)
discriminator_B.apply(weights_init_normal)
```

4.5.2 T-PatchGAN

Per T-PatchGAN il codice applica Xavier, con un opportuno *gain*, che dipende dalla funzione di attivazione che segue il layer:

```
def weights_init_xavier(m):
    classname = m.__class__.__name__
    if classname.find('Conv2d') != -1:
        nn.init.xavier_uniform_(m.weight.data,
         ↪  gain=nn.init.calculate_gain('leaky_relu', 0.2))
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Norm') != -1:  # This will cover
     ↪  BatchNorm, InstanceNorm, and LayerNorm
        if hasattr(m, 'weight') and m.weight is not None:
            nn.init.normal_(m.weight.data, 1, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Linear') != -1:
        nn.init.xavier_uniform_(m.weight.data,
         ↪  gain=nn.init.calculate_gain('relu'))
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)

# Xavier initialization with a gain of 1 for the last two linear
 ↪  layers
def xavier_unit_weights(m):
        torch.nn.init.xavier_uniform_(m.weight, gain=1)
        if hasattr(m, 'bias') and m.bias is not None:
```

```
            nn.init.constant_(m.bias.data, 0.0)


for i, d in enumerate([discriminator_A, discriminator_B]):
    # Apply the weights initialization function to the
     ↪  discriminator
    d.apply(weights_init_xavier)
    d.attention_pool.apply(xavier_unit_weights)
    d.fc.apply(xavier_unit_weights)
```

Il codice per le funzioni di normalizzazione rimane, invece, lo stesso della PatchGAN.

4.6 Metrica KID

Per calcolare la metrica KID, come descritto nella Sezione 3.5.6, abbiamo utilizzato il repository GitHub `https://github.com/toshas/torch-fidelity`. Una volta generate all'interno di una cartella le immagini dei generatori, per ciascun dominio A e B, abbiamo calcolato il KID score tramite la seguente funzione Python:

```
from torch_fidelity import calculate_metrics

kid_subset_size = min(len([f for f in os.listdir(cyclegan_folder_A)
 ↪  if f.endswith('.jpg') or f.endswith('.png')]), len([f for f in
 ↪  os.listdir(generated_A_folder) if f.endswith('.jpg') or
 ↪  f.endswith('.png')]))

# Calculate FID
metrics = calculate_metrics(
    input1=cyclegan_folder_A,
    input2=generated_A_folder,
    isc=False,
    fid=False,
    kid=True,
    kid_subset_size = kid_subset_size,
)
```

`kid_subset_size` indica la dimensione del batch dove applicare l'Equazione 3.5.1. Dal momento che le immagini del batch vengono selezionate casualmente, il processo viene ripetuto di default 100 volte da `torch_fidelity`, producendo, così, in output la media e la deviazione standard dei risultati.

CAPITOLO 5

Esperimenti

Nel capitolo che segue procederemo con un'analisi delle prestazioni della nostra rete avversaria T-PatchGAN, confrontandola direttamente con la PatchGAN 70x70. Per garantire un confronto equo, entrambe le reti sono state configurate con gli stessi generatori e una medesima inizializzazione dei pesi, ottenuta attraverso la fase di pre-training.

5.1 Dataset Utilizzati

Per quanto riguarda i dataset utilizzati nel nostro studio, abbiamo iniziato esaminando alcuni classici impiegati negli studi originali di CycleGAN, come ad esempio la conversione di mappe satellitari in mappe stradali. Successivamente, abbiamo incluso dataset più complessi come *selfie2anime*, utilizzati dagli autori di UVCGAN.

5.2 T-PatchGAN $n \times n$

Chiameremo T-PatchGAN 4×4 la T-PatchGAN che usa patch locali di dimensioni 4×4, e T-PatchGAN 8×8 quella che utilizza patch locali di dimensioni 8×8.

5.3 Dataset Mappe Stradali-Satellitari

Il dataset è diviso nel dominio A delle immagini satellitari e nel dominio B delle immagini stradali, come mostrato in Figura 5.1.

Inoltre, nella Tabella 5.1 è indicato il numero di immagini, per ciascun dominio A e B, per il training e il testing.

5.3.1 Pre-Training

Per il pre-training, abbiamo utilizzato l'algoritmo di ottimizzazione `Adam`, e lo scheduler *Cosine Annealing*, per aggiustare il learning rate nel corso del training.

La dimensione del batch di training è pari a 16.

I dettagli dei parametri sono riassunti nelle Tabelle 5.2 e 5.3

Il pre-training è stato condotto per un totale di 150 epoche, usando la `L1 Loss`. I risultati sono mostrati nelle Figure 5.2 e 5.3.

Figura 5.1: Esempi di immagini del dataset delle mappe Stradali-Satellitari

Cartella	Numero Immagini
trainA	1096
trainB	1096
testA	1098
testB	1098

Tabella 5.1: Numero di immagini del dataset delle mappe Stradali-Satellitari

Parametro	Valore
betas	0.9, 0.99
learning rate (lr)	1e-4
weight_decay	0.05

Tabella 5.2: Parametri di Adam per il Pre-Training di mappe Stradali-Satellitari

Parametro	Valore
T_max	50
eta_min	1e-6

Tabella 5.3: Parametri di Cosine Annealing per il Pre-Training di mappe Stradali-Satellitari

5.3.2 Training

Il training è stato condotto usando `Adam` come otimizzatore, i cui parametri sono riassunti nella Tabella 5.4. Il numero di epoche utilizzate è pari a 50, per un totale di 54.800 iterazioni, mentre la GPU utilizzata è la Tesla V100 di Google.

Per ciò che riguarda la CycleGAN Loss, i valori di λ utilizzati per la Cycle Consistency Loss e l'Identity Loss sono riassunti nella Tabella 5.5.

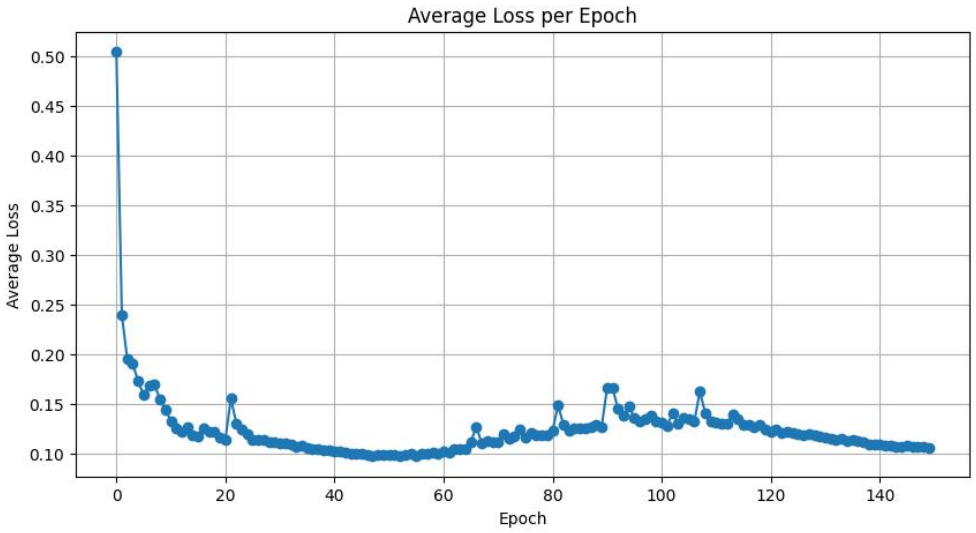

Figura 5.2: Loss Pre-Training delle mappe Stradali-Satellitari

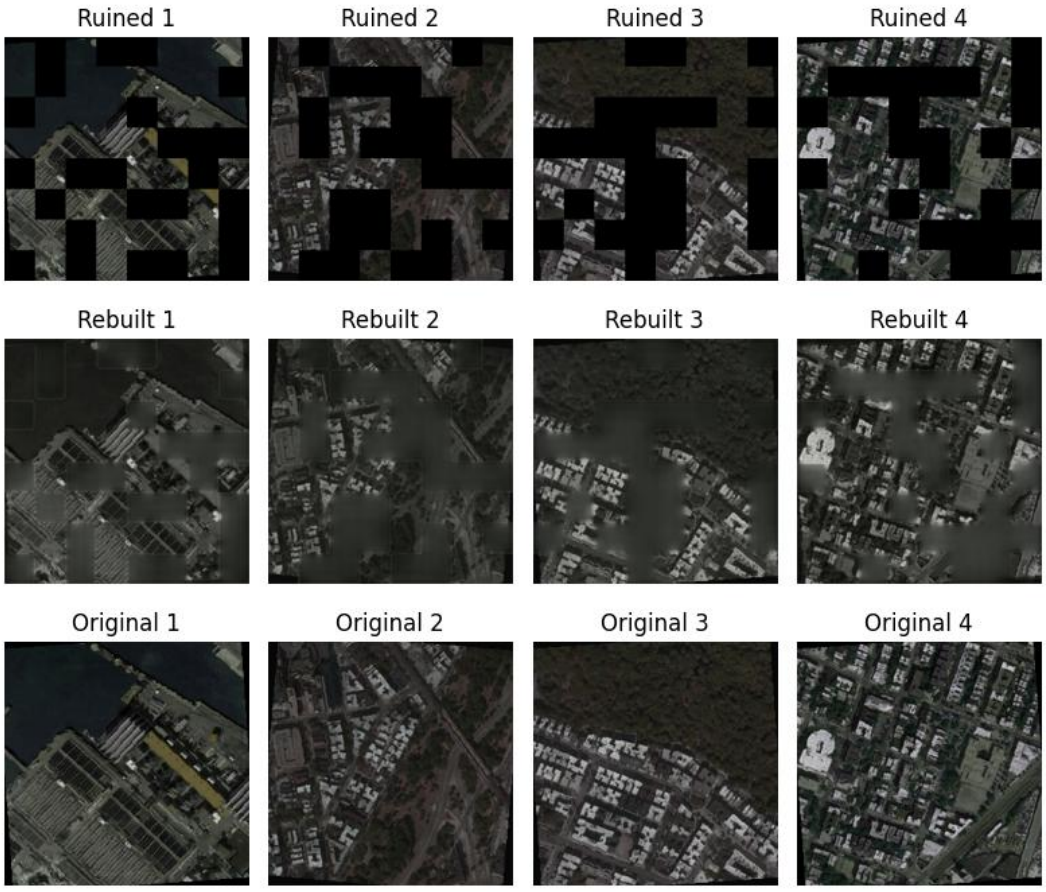

Figura 5.3: Esempi dei risultati di Pre-Training nelle mappe Stradali-Satellitari

Loss

La somma delle loss dei generatori e dei discriminatori è mostrata in Figura 5.4.

In particolare, si evidenzia la capacità di T-PatchGAN di garantire un training stabile, con un andamento simile a quello della PatchGAN.

Parametro	Valore
betas	0.5, 0.99
learning rate (lr)	1e-4

Tabella 5.4: Parametri di Adam per il Training di mappe Stradali-Satellitari

Loss	Valore
Cycle Consistency Loss (λ_{cyc})	10
Identity Loss (λ_{idt})	0

Tabella 5.5: Valori di CycleGAN Loss per mappe Stradali-Satellitari

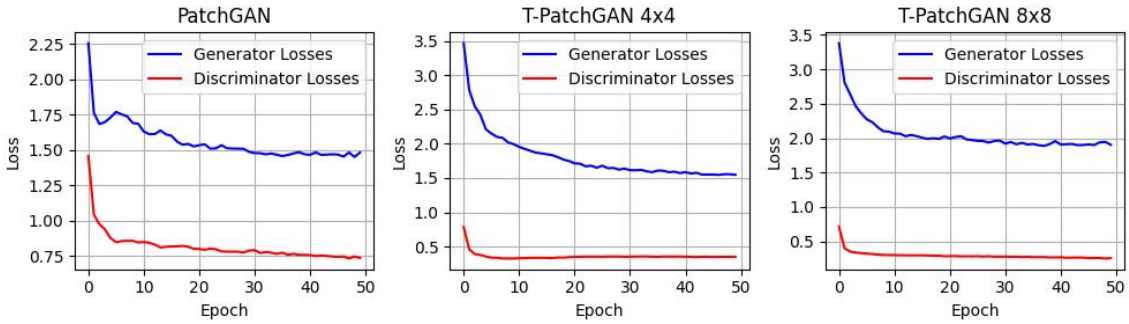

Figura 5.4: Loss Training delle mappe Stradali-Satellitari

Consumo di Risorse

Nella Tabella 5.6 sono evidenziati i tempi di training delle varie reti avversarie, insieme al consumo medio di GPU.

Discriminatore	Minuti medi per epoca	Consumo RAM della GPU
PatchGAN	2.58	4.8 GB
T-PatchGAN 4×4	2.77	4.1 GB
T-PatchGAN 8×8	2.74	4.1 GB

Tabella 5.6: Risorse consumate dalle Reti Avversarie per mappe Stradali-Satellitari

La T-PatchGAN, nonostante occupi quasi 1GB in meno di memoria, ha tempi medi per epoca leggermente superiori alla PatchGAN.

KID Score

Nella Tabella 5.7 sono riportati i vari KID score calcolati con le immagini di test.

La T-PatchGAN 4×4 supera la PatchGAN in termini di KID Score, anche se qualitativamente tutte ottengono buoni risultati alla cinquantesima epoca, come mostrato nelle Figure 5.5 e 5.6.

Discriminatore	KID (×100)	
	Stradali → Satellitari	**Satellitari → Stradali**
PatchGAN	0.051 ± 6.96e-5	0.14 ± 1e-4
T-PatchGAN 4 × 4	**0.049 ± 6.35e-5**	**0.11 ± 1e-4**
T-PatchGAN 8 × 8	0.067 ± 8.06e-5	0.14 ± 1e-4

Tabella 5.7: KID-Score delle mappe Stradali-Satellitari

5.4 Dataset Fotografie-Vangogh

Il dataset è diviso nel dominio A dei quadri di van Gogh, e nel dominio B delle fotografie di paesaggi, come mostrato in Figura 5.7.

Inoltre, nella Tabella 5.8 è indicato il numero di immagini, per ciascun dominio A e B, per il training e il testing.

Cartella	Numero Immagini
trainA	400
trainB	6287
testA	400
testB	751

Tabella 5.8: Numero di immagini del dataset Fotografie-Vangogh

5.4.1 Pre-Training

Per il pre-training, abbiamo utilizzato l'algoritmo di ottimizzazione `Adam` e lo scheduler *Cosine Annealing* per aggiustare il learning rate nel corso del training.

La dimensione del batch di training è pari a 16.

I dettagli dei parametri sono riassunti nelle Tabelle 5.9 e 5.10

Parametro	Valore
betas	0.9, 0.99
learning rate (lr)	1e-4
weight_decay	0.05

Tabella 5.9: Parametri di Adam per il Pre-Training del dataset Fotografie-Vangogh

Il pre-training è stato condotto per un totale di 150 epoche, usando la `L1 Loss`. I risultati sono mostrati nelle Figure 5.8 e 5.9.

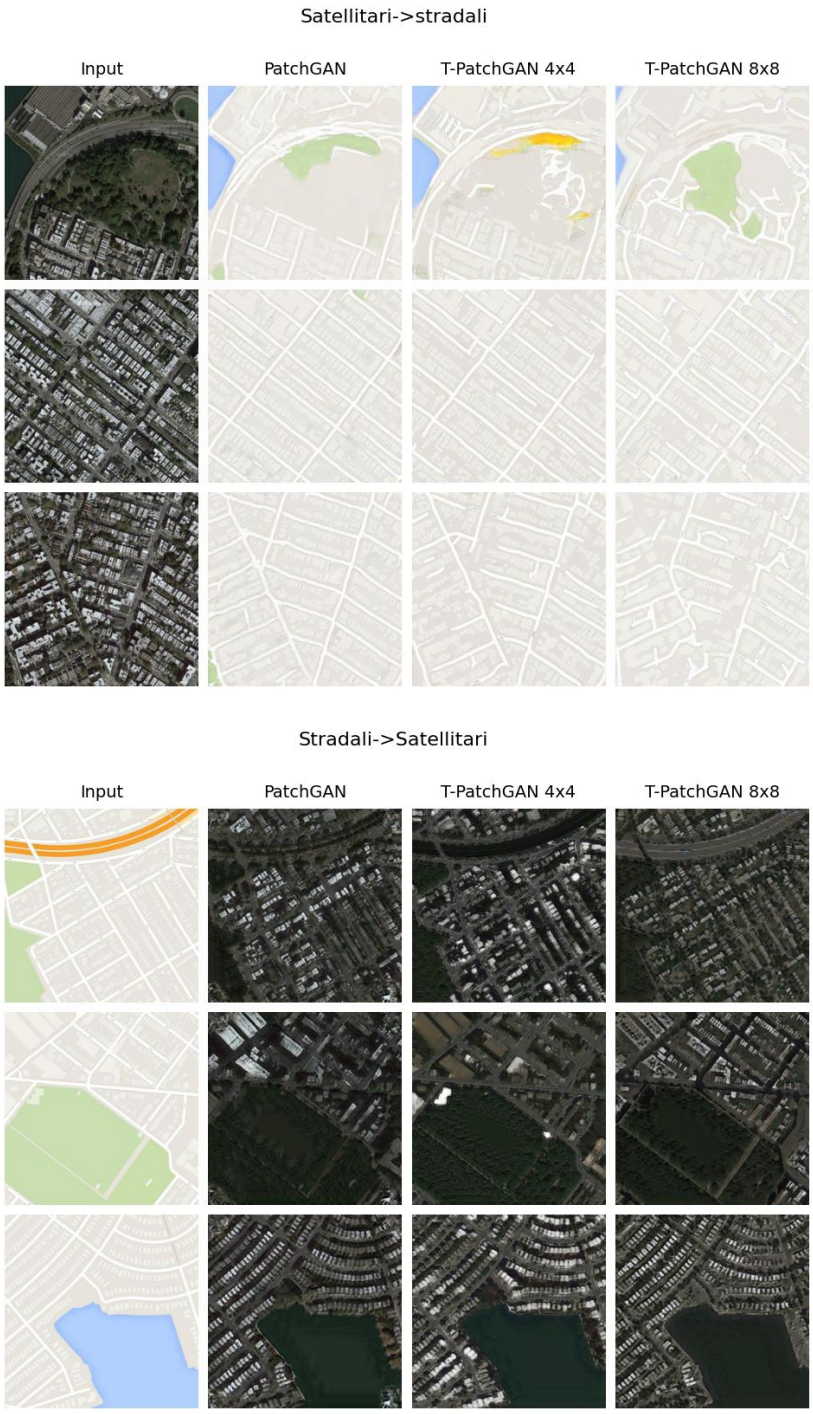

Figura 5.5: Esempi di risultati del dataset mappe Stradali-Satellitari

Parametro	Valore
T_max	50
eta_min	1e-6

Tabella 5.10: Parametri di Cosine Annealing per il Pre-Training di Fotografie-Vangogh

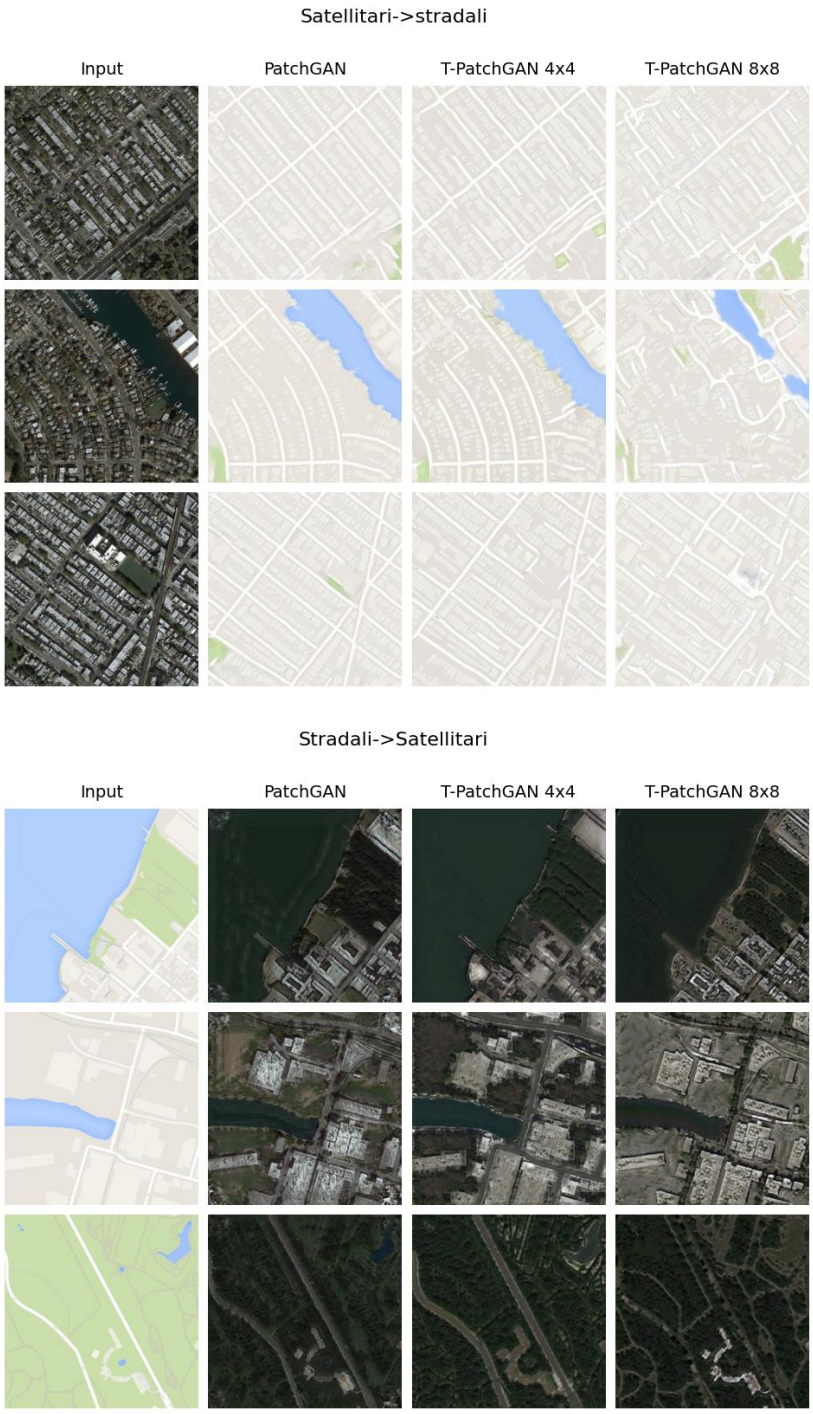

Figura 5.6: Altri esempi di risultati del dataset mappe Stradali-Satellitari

5.4.2 Training

Il training è stato condotto usando `Adam` come otimizzatore, i cui parametri sono riassunti nella Tabella 5.11. Il numero di epoche utilizzate è pari a 8, per un totale di 50.296 iterazioni, mentre la GPU utilizzata è la Tesla V100 di Google.

Per ciò che riguarda la CycleGAN Loss, i valori di λ utilizzati per la Cycle Consistency Loss e l'Identity Loss sono riassunti nella Tabella 5.12.

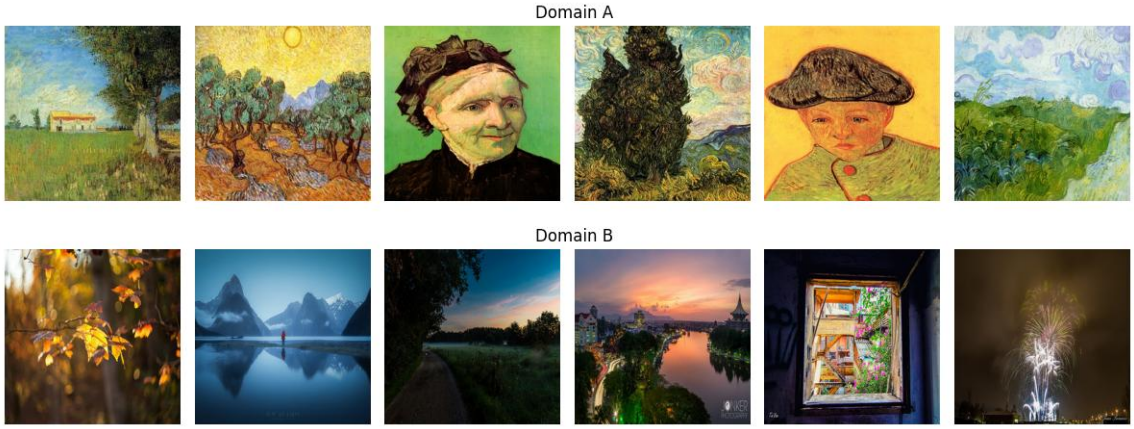

Figura 5.7: Esempi di immagini del dataset Fotografie-Vangogh

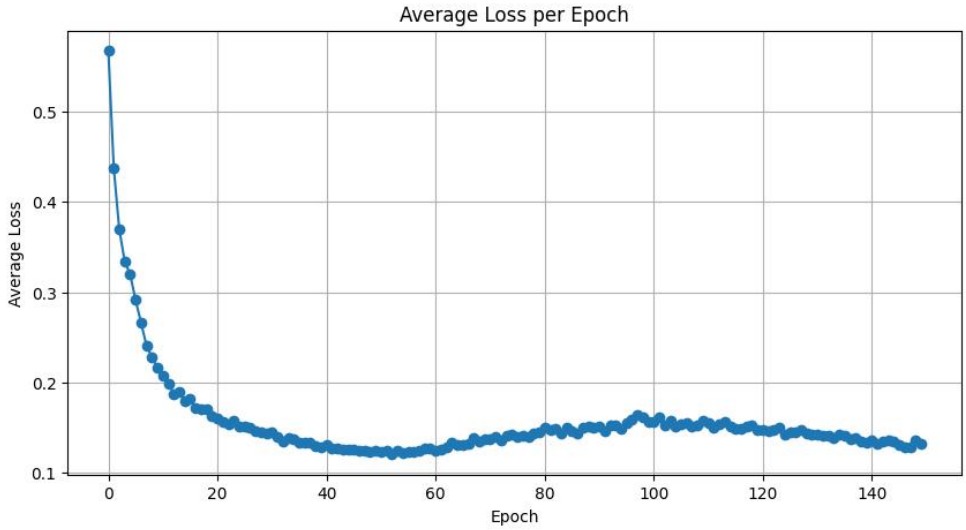

Figura 5.8: Loss Pre-Training del dataset Fotografie-Vangogh

Parametro	**Valore**
betas	0.5, 0.99
learning rate (lr)	1e-4

Tabella 5.11: Parametri di Adam per il Training di Fotografie-Vangogh

Loss	**Valore**
Cycle Consistency Loss (λ_{cyc})	10
Identity Loss (λ_{idt})	5

Tabella 5.12: Valori di CycleGAN Loss per il dataset Fotografie-Vangogh

Loss

La somma delle loss dei generatori e dei discriminatori è mostrata in Figura 5.10.

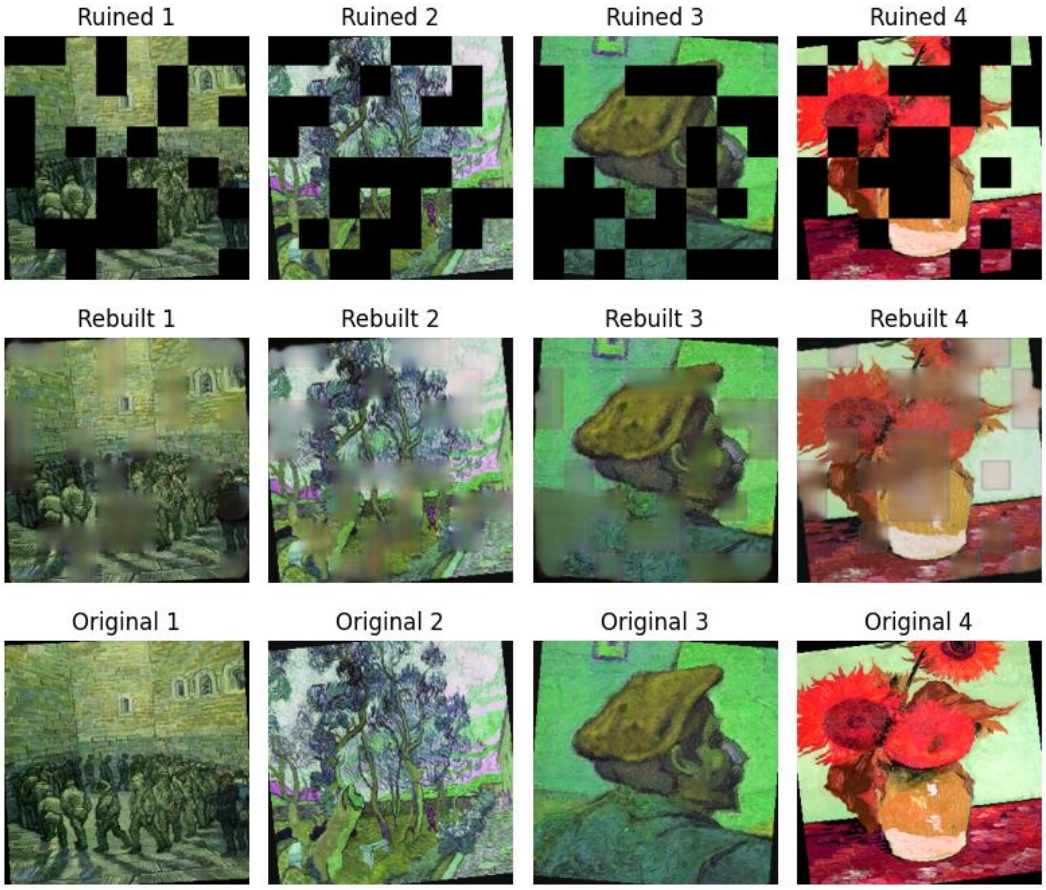

Figura 5.9: Esempi di risultati del Pre-Training del dataset Fotografie-Vangogh

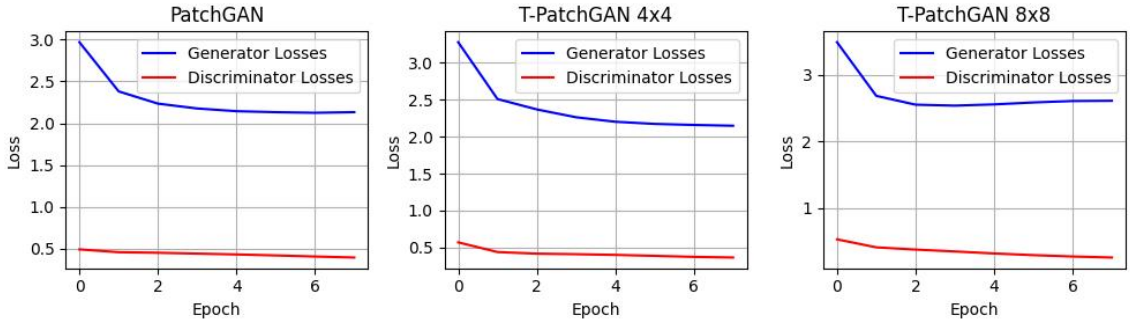

Figura 5.10: Loss Training del dataset Fotografie-Vangogh

Anche in questo caso, si evidenzia la capacità di T-PatchGAN 4×4 di garantire un training stabile; la T-PatchGAN 8×8 presenta, invece, una curva della loss dei generatori che tende a divergere, segnale che potrebbe non esserci un equilibrio ottimale tra le due reti.

Consumo di Risorse

Nella Tabella 5.13 sono evidenziati i tempi di training delle varie reti avversarie, insieme al consumo medio di GPU.

Rispetto al dataset delle mappe Stradali-Satellitari, i tempi delle T-PatchGAN sono stati nettamente inferiori alla PatchGAN; questo potrebbe essere dovuto a una volatilità delle

Discriminatore	Minuti medi per epoca	Consumo RAM della GPU
PatchGAN	21.46	5.6 GB
T-PatchGAN 4×4	19.51	5.0 GB
T-PatchGAN 8×8	19.54	5.0 GB

Tabella 5.13: Risorse consumate dalle Reti Avversarie per il dataset Fotografie-Vangogh

risorse messe a disposizione da Google Colab. Inoltre, il maggior consumo di RAM della GPU è dovuto all'introduzione dell'Identity Loss.

KID Score

Nella Tabella 5.14 sono riportati i vari KID score calcolati con le immagini di test.

Discriminatore	KID (×100)	
	Fotografie → Vangogh	**Vangogh → Fotografie**
PatchGAN	0.053 ± 1e-3	0.098 ± 1e-3
T-PatchGAN 4×4	0.046 ± 1e-3	0.083 ± 1e-3
T-PatchGAN 8×8	**0.034 ± 1e-3**	**0.077 ± 1e-3**

Tabella 5.14: KID-Score del dataset Fotografie-Vangogh

In questo caso, entrambe le T-PatchGAN superano la PatchGAN in termini di KID Score. Alcuni esempi di risultati sono mostrati nelle Figure 5.11 e 5.12.

Dall'analisi dei risultati, emerge che la T-PatchGAN con configurazione 4×4 fornisce immagini dal realismo superiore, sebbene presenti un KID Score leggermente più alto rispetto alla Versione 8×8. È interessante notare che la T-PatchGAN 8×8 tende a generare artefatti nelle immagini; questo problema, invece, non si verifica nella configurazione 4×4.

5.5 Dataset Selfie-Anime

Il dataset è diviso nel dominio A dei selfie e nel dominio B degli anime, come mostrato in Figura 5.13.

Inoltre, nella Tabella 5.15 è indicato il numero di immagini, per ciascun dominio A e B, per il training e il testing.

5.5.1 Pre-Training

Per il pre-training, abbiamo utilizzato l'algoritmo di ottimizzazione `Adam`, e lo scheduler *Cosine Annealing* per aggiustare il learning rate nel corso del training.

La dimensione del batch di training è pari a 16.

I dettagli dei parametri sono riassunti nelle Tabelle 5.16 e 5.17

Il pre-training è stato condotto per un totale di 150 epoche, usando la `L1 Loss`. I risultati sono mostrati nelle Figure 5.14 e 5.15.

Figura 5.11: Esempi di risultati del dataset Fotografie-Vangogh

5.5.2 Training

Il training è stato condotto usando `Adam` come otimizzatore, i cui parametri sono riassunti nella Tabella 5.18. Il numero di epoche utilizzate è pari a 20, per un totale di 68.000 iterazioni, mentre la GPU utilizzata è la Tesla V100 di Google.

Per ciò che riguarda la CycleGAN Loss, i valori di λ utilizzati per la Cycle Consistency Loss e l'Identity Loss sono riassunti nella Tabella 5.19.

Figura 5.12: Altri esempi di risultati del dataset Fotografie-Vangogh

Loss

La somma delle loss dei generatori e dei discriminatori è mostrata in Figura 5.16.

Consumo di Risorse

Nella Tabella 5.20 sono evidenziati i tempi di training delle varie reti avversarie, insieme al consumo medio di GPU.

Anche in questo caso, i tempi medi del training somo simili.

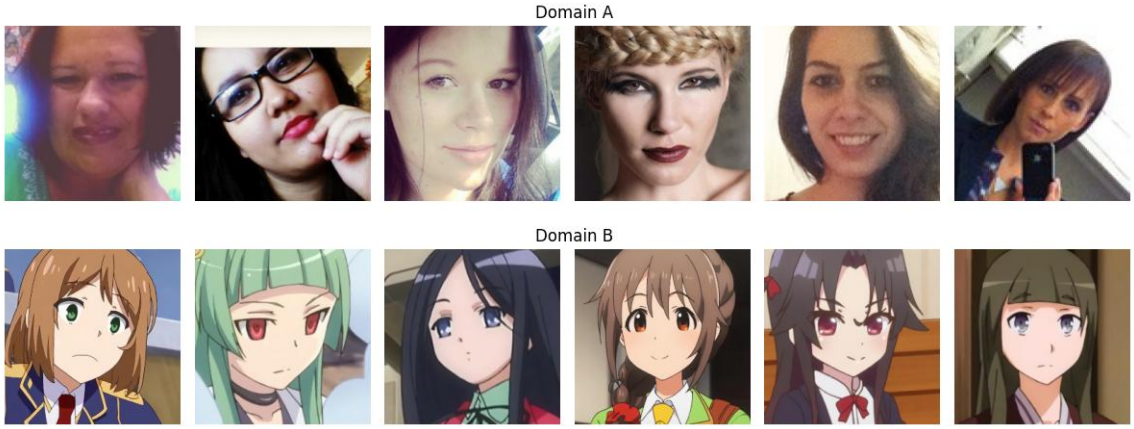

Figura 5.13: Esempi di immagini del dataset Selfie-Anime

Cartella	Numero Immagini
trainA	3400
trainB	2610
testA	100
testB	100

Tabella 5.15: Numero di immagini del dataset Selfie-Anime

Parametro	Valore
betas	0.9, 0.99
learning rate (lr)	1e-4
weight_decay	0.05

Tabella 5.16: Parametri di Adam per il Pre-Training del dataset Selfie-Anime

Parametro	Valore
T_max	50
eta_min	1e-6

Tabella 5.17: Parametri di Cosine Annealing per il Pre-Training del dataset Selfie-Anime

KID Score

Nella Tabella 5.21 sono riportati i vari KID score calcolati con le immagini di test.

L'analisi delle Figure 5.17 e 5.18 rivela una limitazione significativa della T-PatchGAN, ovvero la sua difficoltà nel catturare pattern globali. Questa caratteristica si manifesta nella sua incapacità di trasformare efficacemente un selfie in uno stile anime. La T-PatchGAN, concentrandosi prevalentemente sulle texture locali, non riesce a realizzare la conversione in

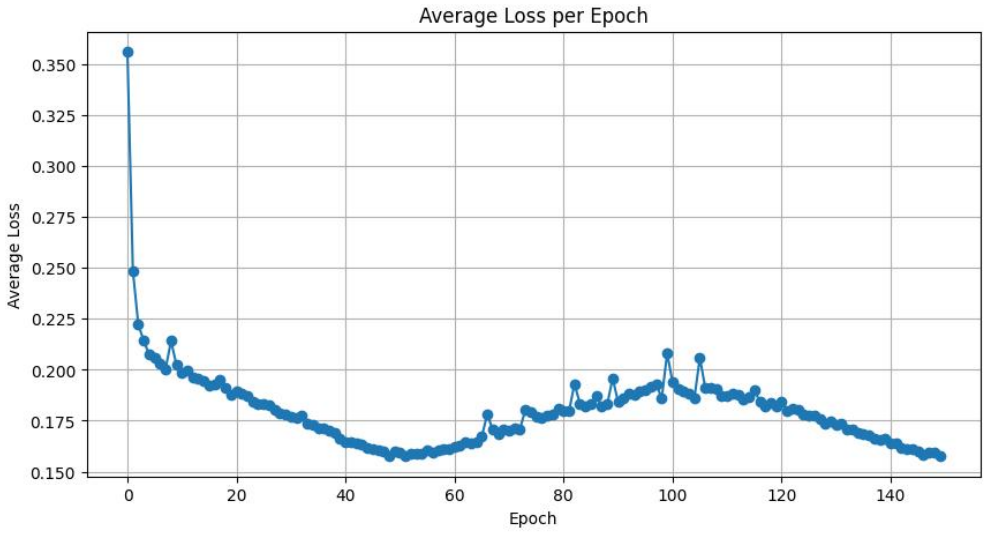

Figura 5.14: Loss Pre-Training per il dataset Selfie-Anime

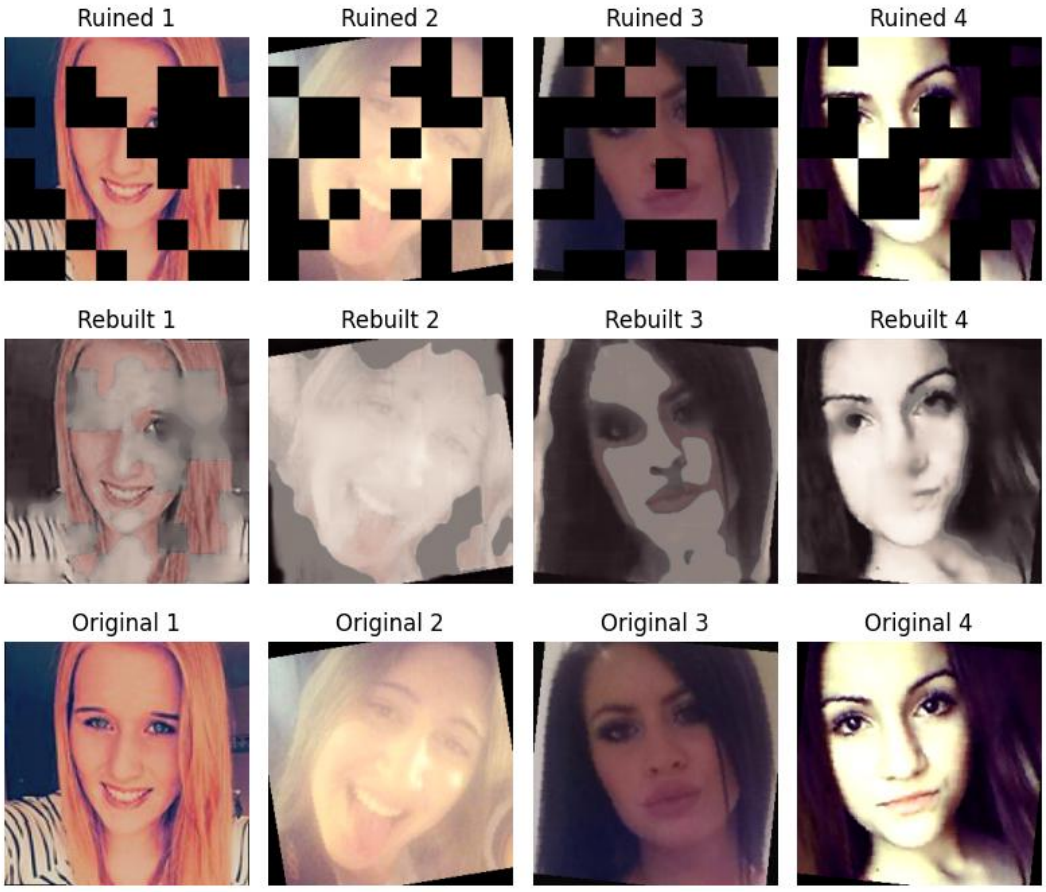

Figura 5.15: Esempi dei risultati di Pre-Training del dataset Selfie-Anime

modo soddisfacente, a differenza della PatchGAN che gestisce questa trasformazione con maggiore efficacia.

Nonostante questo limite, è interessante notare che il KID Score per la conversione da Selfie ad Anime è migliore nelle T-PatchGAN. Questo risultato è dovuto al fatto che le feature estratte dalle immagini generate dalla T-PatchGAN risultano essere più vicine alle mappe

Parametro	Valore
betas	0.5, 0.99
learning rate (lr)	1e-4

Tabella 5.18: Parametri di Adam per il Training del dataset Selfie-Anime

Loss	Valore
Cycle Consistency Loss (λ_{cyc})	10
Identity Loss (λ_{idt})	0

Tabella 5.19: Valori di CycleGAN Loss per il dataset Selfie-Anime

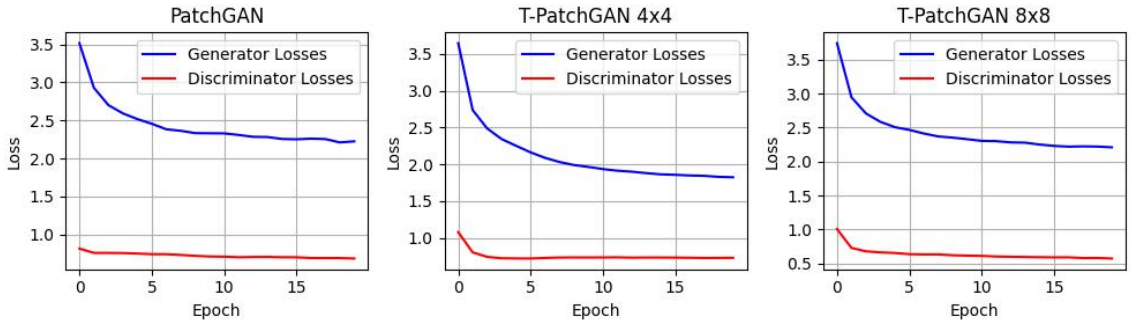

Figura 5.16: Loss Training del dataset Selfie-Anime

Discriminatore	Minuti medi per epoca	Consumo RAM della GPU
PatchGAN	8.40	4.8 GB
T-PatchGAN 4×4	8.17	4.1 GB
T-PatchGAN 8×8	8.91	4.1 GB

Tabella 5.20: Risorse consumate dalle Reti Avversarie per il dataset Selfie-Anime

Discriminatore	KID (×100)	
	Anime → Selfie	Selfie → Anime
PatchGAN	**0.070 ± 4e-3**	0.077 ± 4e-3
T-PatchGAN 4×4	0.097 ± 5e-3	**0.053 ± 3e-3**
T-PatchGAN 8×8	0.098 ± 6e-3	0.071 ± 2e-3

Tabella 5.21: KID-Score del dataset Selfie-Anime

di feature estratte dai dati di training degli Anime, se confrontate con quelle estratte dalla PatchGAN.

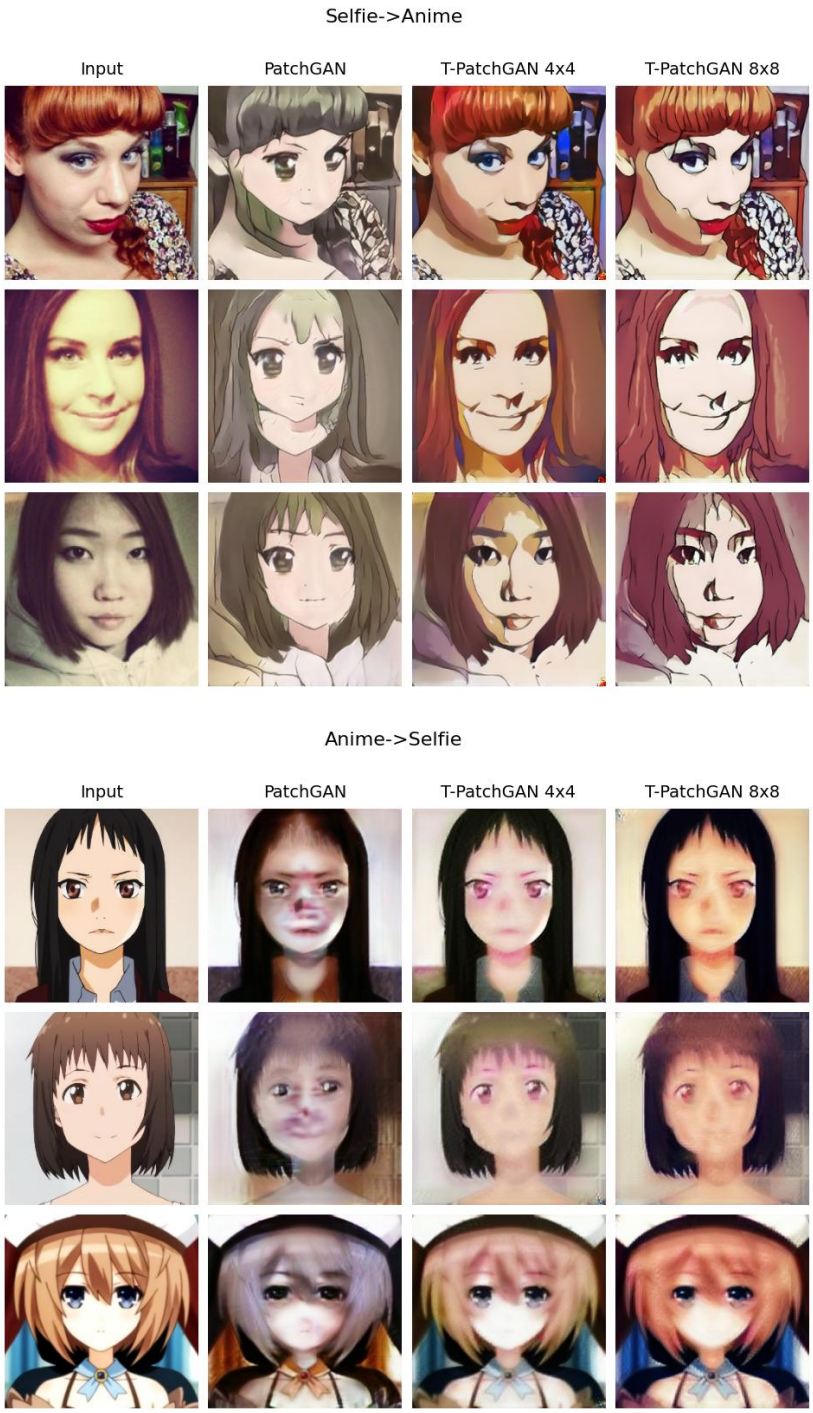

Figura 5.17: Esempi di risultati del dataset Selfie-Anime

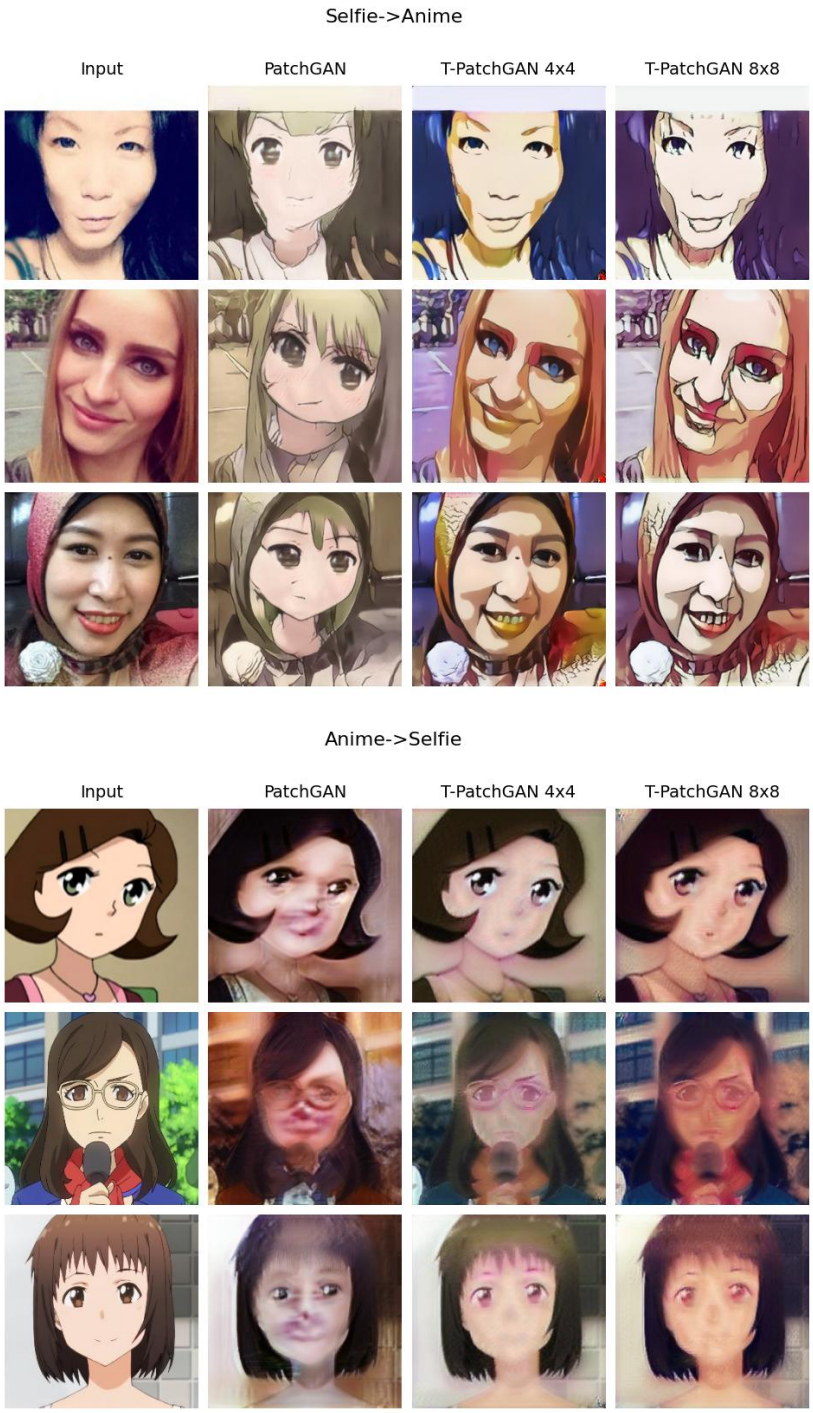

Figura 5.18: Altri esempi di risultati del dataset Selfie-Anime

CAPITOLO 6

Possibili miglioramenti dell'architettura proposta

In questo capitolo si cercherà di proporre un possibile miglioramento della T-PatchGAN, al fine di consentire di catturare correttamente pattern globali, come nel caso del dataset Selfie-Anime, e procedere così con la definizione di possibili obiettivi futuri.

6.1 TGlobal-PatchGAN

Nel contesto della traduzione tra domini sufficientemente diversi, la T-PatchGAN mostra limitazioni dovute alla sua focalizzazione sulle caratteristiche "locali". Per superare questa sfida, introduciamo la TGlobal-PatchGAN, una variante che mira a integrare l'efficienza computazionale e la qualità della T-PatchGAN — superiore alla PatchGAN per le texture locali — con la capacità della PatchGAN di identificare pattern globali complessi.

6.1.1 Architettura Proposta

L'architettura della TGlobal-PatchGAN, illustrata in Figura 6.1, è simile a quella della T-PatchGAN. La novità principale risiede nell'aggiunta di un blocco *Dilated Conv* all'output. Questo blocco elabora l'output dei primi due strati convoluzionali applicando una convoluzione 2D con parametri $out\_channels = 256$, $stride = 2$, $padding = 1$, e $dilated = 4$. Quest'ultimo parametro espande il kernel a 13×13, mantenendo, però, i parametri allenabili a $4 \times 4 = 16$ per filtro, come mostrato in Figura 6.2. Tale configurazione permette di ampliare il receptive field mantenendo una ridotta complessità computazionale. Un ulteriore strato convoluzionale 2D con $kernel\_size = 4$ genera, infine, la mappa di output, analogamente a quanto avviene nella PatchGAN, i cui pixel verranno classificati come 0 (Fake) e 1 (Real).

Utilizzando l'Equazione 2.1.3, possiamo calcolare che il receptive field per i pixel della feature map estratta dal *Dilated Conv Block* è 82×82.

Questa combinazione di mappe *locali* dal Transformer Encoder e *globali* dai blocchi convoluzionali consente quindi di integrare gli approcci della PatchGAN e della T-PatchGAN, mantenendo un'efficiente complessità computazionale.

Inoltre, abbiamo apportato modifiche al Transformer Encoder, che ora include un unico layer anziché due stacked layer.

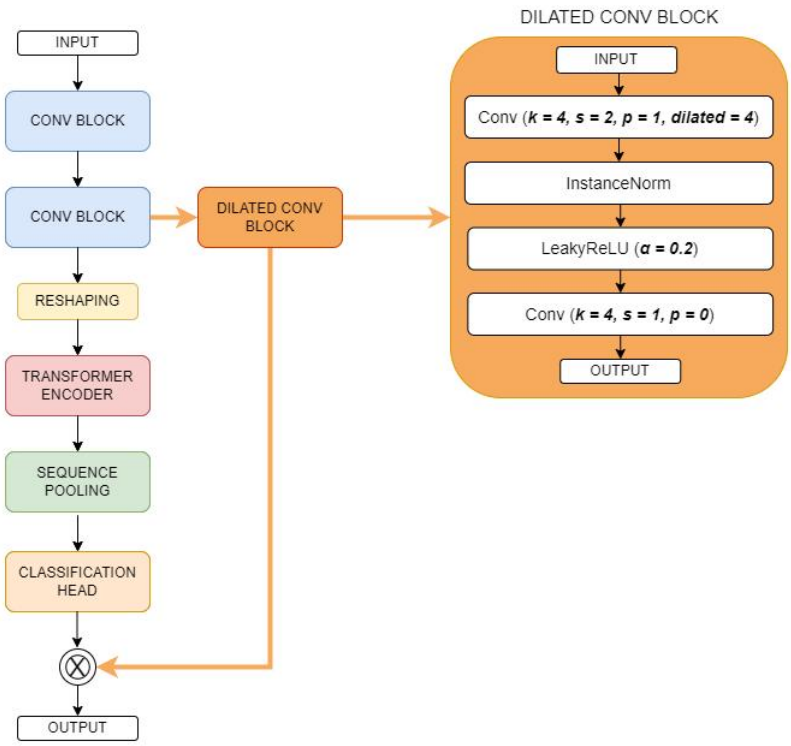

Figura 6.1: Architettura della TGlobal-PatchGAN

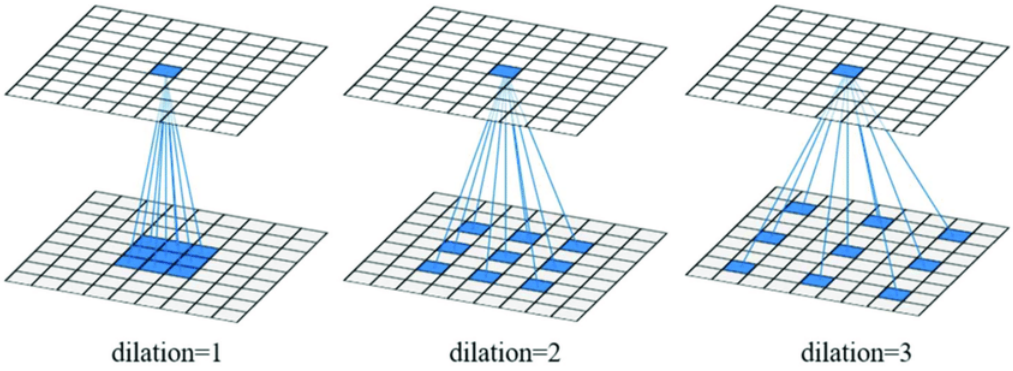

Figura 6.2: Esempio di Dilated Convolution

6.1.2 Numero Parametri

Il numero di parametri allenabili della TGlobal-PatchGAN 4×4 è pari a 863.555, contro i 533.186 della T-PatchGAN 4×4, e i 2.766.529 della PatchGAN 70×70.

6.1.3 Implementazione Python

Il codice della nuova architettura è il seguente:

```
class DilatedConvBlock(nn.Module):
    """
    This layer applies a dilated convolutional operation.

    Args:
```

```
        input_channels (int): Number of input channels in the
         ↪ feature map.
        output_channels (int): Number of output channels of the
         ↪ initial feature map.
    """
    def __init__(self, input_channels, output_channels):
        super().__init__()

        # Initial Convolutional Layers
        self.conv_layers = nn.Sequential(
            nn.Conv2d(input_channels, output_channels,
             ↪ kernel_size=4, dilation=4, stride=2, padding=1),
            nn.InstanceNorm2d(output_channels),
            nn.LeakyReLU(0.2),
        )

        self.final_conv = nn.Conv2d(output_channels, 1,
         ↪ kernel_size=4)

    def forward(self, feature_map):
        """
        Forward pass through the network.

        Args:
            feature_map (torch.Tensor): Input feature map of shape
             ↪ [Batch_Size, Channels, Height, Width].

        Returns:
            torch.Tensor: Output tensor of shape [Batch_Size,
             ↪ Height * Width, 1].
        """

        x = self.conv_layers(feature_map)
        x = self.final_conv(x)
        return x.flatten(1).unsqueeze(-1)



class Discriminator(nn.Module):
    def __init__(self, img_size: int = 256, patch_size: int = 4,
     ↪ channels: int = 3, transformer_layers: int = 1, heads: int
     ↪ = 2):
        """
        Discriminator constructor.

        Args:
        img_size (int, optional): The size of the input image.
         ↪ Default is 256.
```

```
        patch_size (int, optional): The size of each image patch
         ↪  for the Transformer. Default is 4.
        channels (int, optional): The number of channels in the
         ↪  input image. Default is 3.
        transformer_layers (int, optional): The number of
         ↪  Transformer encoder layers. Default is 2.
        heads (int, optional): The number of attention heads in the
         ↪  Transformer encoder. Default is 4.
        """
        super().__init__()

        self.patch_size = patch_size
        self.num_patches = (img_size // 4 // patch_size) ** 2
        self.embed_dim = 128

        # Initial Convolutional Layers
        self.conv_layers = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=4, stride=2,
             ↪  padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2)
        )

        # Transformer Configuration
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=self.embed_dim,
            nhead=heads,
            dim_feedforward=self.embed_dim * 4,
            dropout=0.0,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )

        self.transformer_encoder =
         ↪  nn.TransformerEncoder(encoder_layer,
         ↪  num_layers=transformer_layers)

        self.dilated_conv_block = DilatedConvBlock(128, 256)

        # Trainable Positional Embedding
        self.positional_embedding = nn.Parameter(torch.zeros(1,
         ↪  self.patch_size ** 2, self.embed_dim))

        # Attention Pooling Layer
        self.attention_pool = nn.Linear(self.embed_dim, 1)

        # Fully Connected Layer
```

```
        self.fc = nn.Linear(self.embed_dim, 1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Forward pass through the network.

        Args:
        x (torch.Tensor): Input tensor of shape [batch_size,
         ↪  channels, height, width].

        Returns:
        torch.Tensor: Output tensor.
        """
        # Pass through Conv layers
        x = self.conv_layers(x)

        # Apply DilatedConvBlock
        dilated_conv_output = self.dilated_conv_block(x)

        # Shape variables
        b, c, h, w = x.size()

        # Create non-overlapping patches
        x = x.unfold(2, self.patch_size, self.patch_size).unfold(3,
         ↪  self.patch_size, self.patch_size)

        # Reorder dimensions and reshape to get local patches as
         ↪  tokens
        x = x.permute(0, 2, 3, 4, 5, 1).contiguous().view(-1,
         ↪  self.patch_size ** 2, self.embed_dim)

        x = x + self.positional_embedding

        # Pass through Transformer encoder
        x = self.transformer_encoder(x)  # Output shape:
         ↪  [batch_size * num_patches, patch_size ** 2, embed_dim]


        # Sequence Pooling
        attn_weights = self.attention_pool(x).squeeze(-1)  # Output
         ↪  shape: [batch_size * num_patches, 1 + patch_size ** 2]
        x = torch.einsum('b n, b n d -> b d',
         ↪  attn_weights.softmax(dim=1), x)  # Output shape:
         ↪  [batch_size * num_patches, embed_dim]

        # Reshape for final layer
        x = x.view(b, self.num_patches, -1)  # Output shape:
         ↪  [batch_size, num_patches, embed_dim]
```

```
        # Pass through Fully Connected layer
        x = self.fc(x)  # Output shape: [batch_size, num_patches,
         ↪  1]

        return torch.cat((x, dilated_conv_output),dim=1)
```

L'inizializzazione dei pesi rimane con Xavier:

```
def weights_init_xavier(m):
    classname = m.__class__.__name__
    if classname.find('Conv2d') != -1:
        nn.init.xavier_uniform_(m.weight.data,
         ↪  gain=nn.init.calculate_gain('leaky_relu', 0.2))
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Norm') != -1:  # This will cover
     ↪  BatchNorm, InstanceNorm, and LayerNorm
        if hasattr(m, 'weight') and m.weight is not None:
            nn.init.normal_(m.weight.data, 1, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Linear') != -1:
        nn.init.xavier_uniform_(m.weight.data,
         ↪  gain=nn.init.calculate_gain('relu'))
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)

# Xavier initialization with a gain of 1 for the last two linear
 ↪  layers
def xavier_unit_weights(m):
        torch.nn.init.xavier_uniform_(m.weight, gain=1)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)



def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find('Conv2d') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Norm') != -1:  # This will cover
     ↪  BatchNorm, InstanceNorm, and LayerNorm
        if hasattr(m, 'weight') and m.weight is not None:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
    elif classname.find('Linear') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
```

```
            nn.init.constant_(m.bias.data, 0.0)


for i, d in enumerate([discriminator_A, discriminator_B]):
    # Apply the weights initialization function to the
     ↪  discriminator
    d.apply(weights_init_xavier)
    d.attention_pool.apply(xavier_unit_weights)
    d.fc.apply(xavier_unit_weights)
    d.dilated_conv_block.final_conv.apply(xavier_unit_weights)
```

6.1.4 Risultati su Dataset Selfie-Anime

Riprendendo il dataset della Sezione 5.5, confronteremo di nuovo le loss, il consumo di risorse e il KID Score delle varie architetture.

Useremo patch locali di dimensioni 4×4 nella TGlobal-PatchGAN, dal momento che questa dimensione dei patch si è rilevata migliore della 8×8 nei precedenti esperimenti.

Loss

Come mostrato in Figura 6.3, la loss della TGlobal-PatchGAN 4×4 risulta molto stabile.

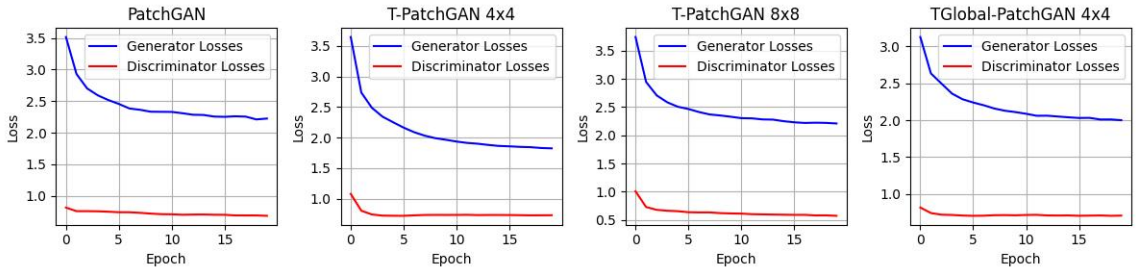

Figura 6.3: Loss Training Selfie-Anime con TGlobal-PatchGAN

Consumo Risorse

Nella Tabella 6.1 si può notare l'elevata efficienza computazionale della TGlobal-PatchGAN a confronto della PatchGAN e delle altre architetture. Questo, ancora una volta, potrebbe essere dovuto a un'instabilità nelle risorse messe a disposizione da Google Colab.

Discriminatore	Minuti medi per epoca	Consumo RAM della GPU
PatchGAN	8.40	4.8 GB
T-PatchGAN 4×4	8.17	4.1 GB
T-PatchGAN 8×8	8.91	4.1 GB
TGlobal-PatchGAN 4×4	7.93	4.3 GB

Tabella 6.1: Risorse consumate dalle Reti Avversarie per il dataset Selfie-Anime con TGlobal-PatchGAN

KID Score

I KID Score sono riassunti nella Tabella 6.2.

Discriminatore	**KID (×100)**	
	Anime → Selfie	**Selfie → Anime**
PatchGAN	0.070 ± 4e-3	0.077 ± 4e-3
T-PatchGAN 4 × 4	0.097 ± 5e-3	**0.053 ± 3e-3**
T-PatchGAN 8 × 8	0.098 ± 6e-3	0.071 ± 2e-3
TGlobal-PatchGAN 4 × 4	**0.065 ± 4e-3**	0.067 ± 3e-3

Tabella 6.2: KID-Score del dataset Selfie-Anime con TGlobal-PatchGAN

La TGlobal-PatchGAN 4 × 4 non solo mostra un KID migliore rispetto alla PatchGAN, ma riesce anche a identificare pattern globali complessi, una capacità che mancava nella T-PatchGAN. Questi miglioramenti sono visibili nelle Figure 6.4 e 6.5, dove si possono osservare le differenze sostanziali nelle prestazioni delle architetture.

6.2 Sviluppi Futuri

La TGlobal-PatchGAN mostra un notevole passo avanti rispetto all'idea originariamente proposta della T-PatchGAN. Questo avanzamento apre nuove prospettive per ulteriori indagini e sperimentazioni. Un aspetto cruciale da esplorare è l'efficacia della TGlobal-PatchGAN su una vasta gamma di dataset. L'impiego di una gamma diversificata di dataset garantirebbe una valutazione più completa delle sue capacità e limitazioni.

Inoltre, è fondamentale condurre test approfonditi con un numero di epoche sufficientemente elevato. Questo approccio permetterebbe di osservare come la rete si evolve e si adatta nel corso del training, fornendo dati preziosi sul suo comportamento nel lungo termine.

Un altro ambito di ricerca importante sarebbe l'uso di generatori diversi dall'UVCGAN, in modo tale da poter scoprire nuove sinergie e modi in cui la TGlobal-PatchGAN può essere ottimizzata o modificata per adattarsi a diversi tipi di generatori.

In conclusione, la T-PatchGAN e la TGlobal-PatchGAN lasciano aperte nuove evoluzioni nel campo delle reti avversarie nel dominio dell'Unpaired Image-to-Image Translation.

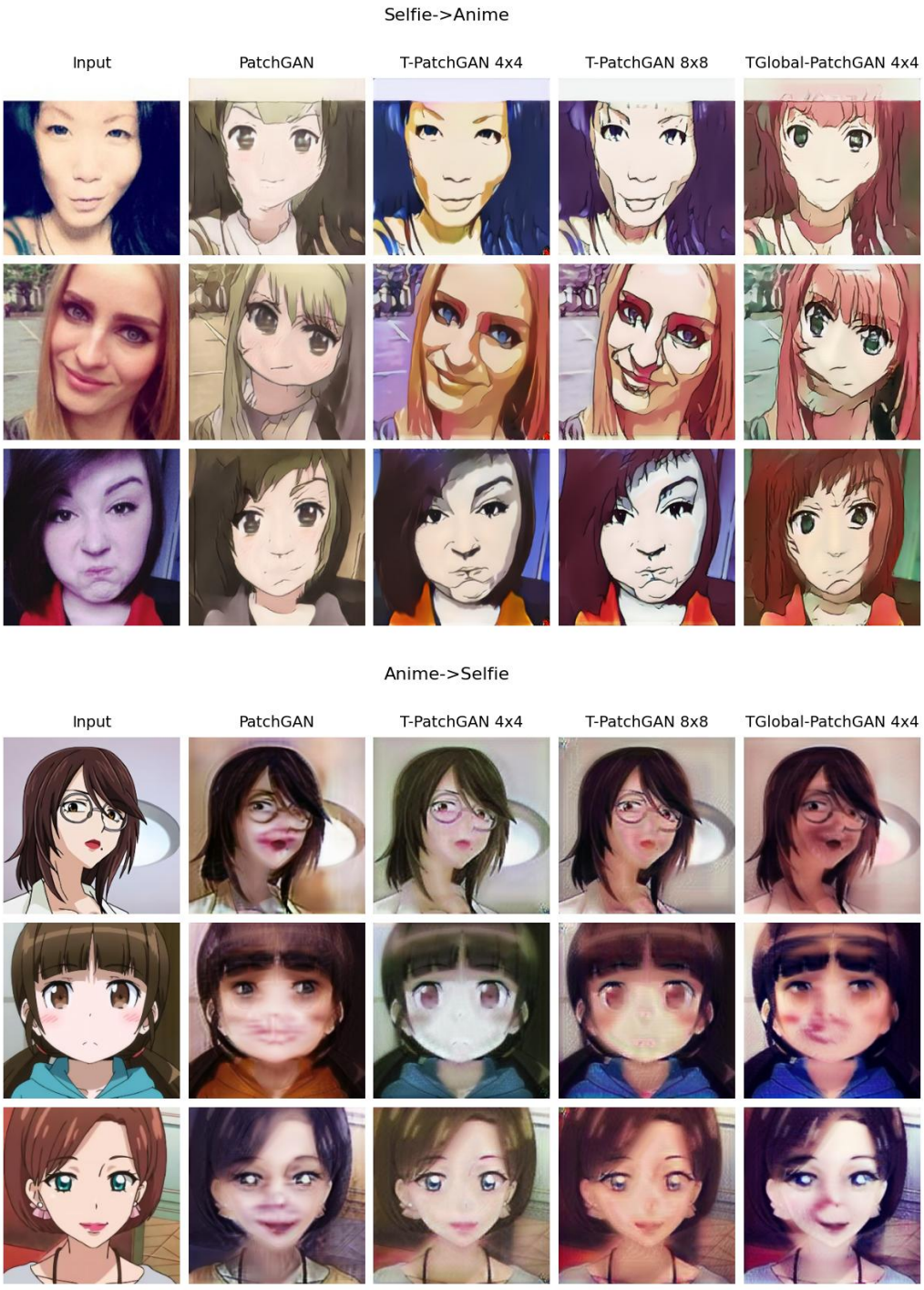

Figura 6.4: Esempi di risultati del dataset Selfie-Anime con TGlobal-PatchGAN

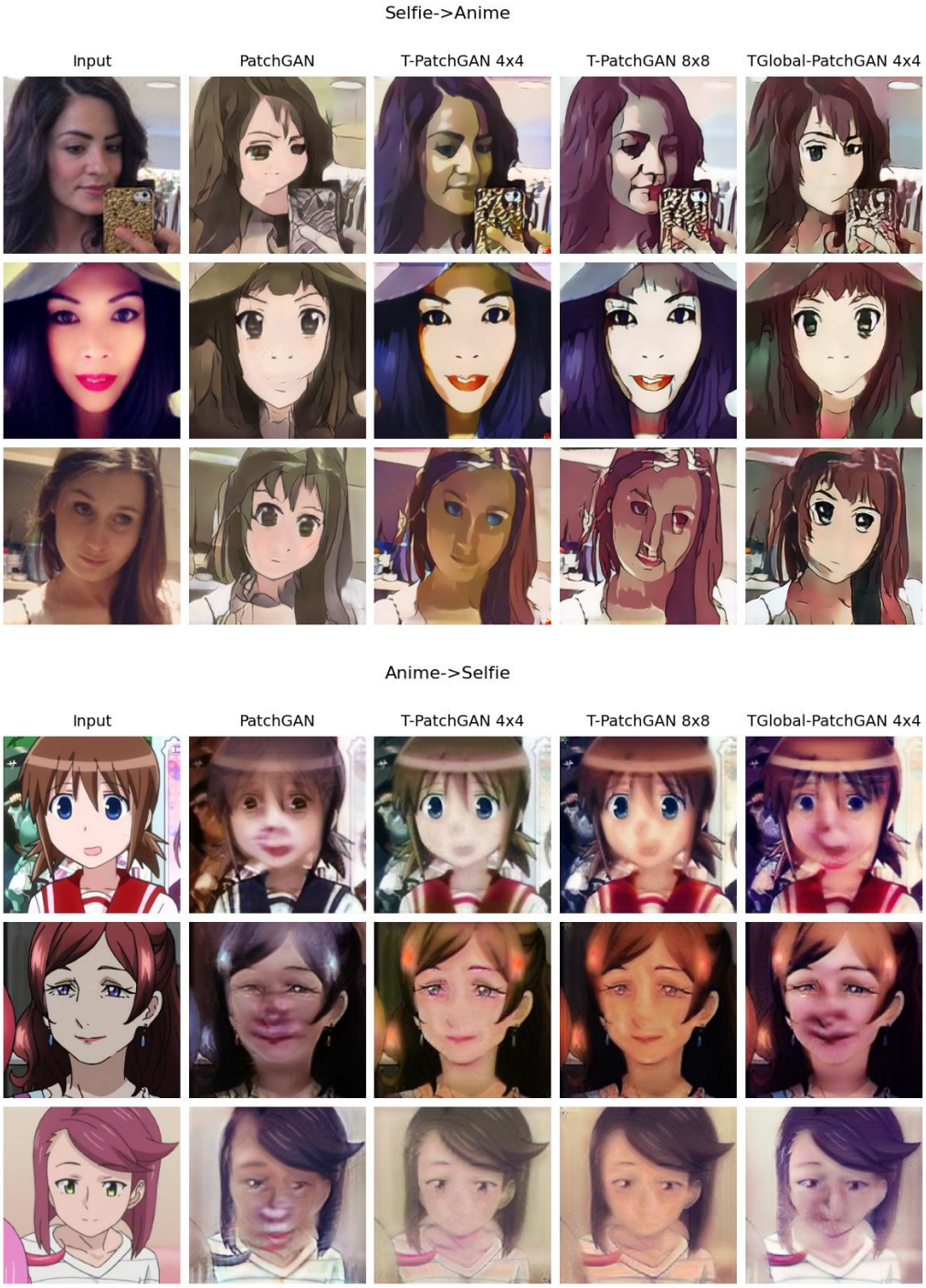

Figura 6.5: Altri esempi di risultati del dataset Selfie-Anime con TGlobal-PatchGAN

Conclusioni

In questo studio siamo partiti dal vasto campo dell'Image-to-Image Translation, giungendo infine all'area più specifica dell'Unpaired Image Translation. Quest'ultimo settore rappresenta una sfida significativa, data la natura non accoppiata dei dati. Il focus principale è stato su come le diverse scelte di loss function, nonché le architetture dei generatori e dei discriminatori, influenzano in modo critico i risultati finali. Queste scelte richiedono un impegno notevole sia in termini di progettazione che di risorse computazionali, considerando l'alto costo di training richiesto per ottenere risultati di qualità.

Uno degli approcci più promettenti in questo dominio è rappresentato dalla PatchGAN, impiegata come rete avversaria. Dal 2017, questa rete si è dimostrata estremamente versatile, adattandosi efficacemente a vari tipi di generatori.

Partendo dalla PatchGAN 70×70, abbiamo sviluppato una nuova rete, denominata T-PatchGAN, particolarmente efficace nell'elaborare le feature locali e che si adatta bene in contesti dove non è richiesta un'ampia capacità di catturare pattern globali. Esempi di tali contesti includono le traslazioni da mappe stradali a satellitari e il dataset Fotografie-Vangogh. Tuttavia, abbiamo identificato delle limitazioni in contesti più complessi come il dataset Selfie-Anime. Per superare queste limitazioni, abbiamo introdotto una variante chiamata TGlobal-PatchGAN, che sfrutta il concetto di rete siamese per sviluppare una capacità parallela, attraverso layer convoluzionali, di catturare anche le relazioni globali, oltre che quelle locali. I risultati preliminari sono promettenti e suggeriscono che la TGlobal-PatchGAN potrebbe essere un ottimo punto di partenza per ulteriori raffinamenti.

In ottica di obiettivi futuri, l'attenzione si concentra sul perfezionamento delle TGlobal-PatchGAN e T-PatchGAN, mirando a migliorarne le prestazioni sia qualitativamente che computazionalmente. Ciò include condurre test estesi su dataset complessi e il possibile sviluppo di un generatore più integrato con queste reti avversarie, per massimizzare l'efficacia e l'efficienza del processo di training.

Bibliografia

BIŃKOWSKI, M., SUTHERLAND, D. J., ARBEL, M. e GRETTON, A. (2018), «Demystifying MMD GANs», *ArXiv*, URL https://arxiv.org/abs/1801.01401. (Cited at page 30)

CARION, N., MASSA, F., SYNNAEVE, G., USUNIER, N., KIRILLOV, A. e ZAGORUYKO, S. (2020), «End-to-End Object Detection with Transformers», *ArXiv*, URL https://arxiv.org/abs/2005.12872. (Cited at page 8)

CHEN, C.-F., FAN, Q. e PANDA, R. (2021a), «CrossViT: Cross-Attention Multi-Scale Vision Transformer for Image Classification», *ArXiv*, URL https://arxiv.org/abs/2103.14899. (Cited at page 6)

CHEN, J., LU, Y., YU, Q., LUO, X., ADELI, E., WANG, Y., LU, L., YUILLE, A. L. e ZHOU, Y. (2021b), «TransUNet: Transformers Make Strong Encoders for Medical Image Segmentation», *ArXiv*, URL https://arxiv.org/abs/2102.04306. (Cited at page 8)

DEMIR, U. e UNAL, G. (2018), «Patch-Based Image Inpainting with Generative Adversarial Networks», *ArXiv*, URL https://arxiv.org/abs/1803.07422. (Cited at page 13)

DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J. e HOULSBY, N. (2020), «AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE», *ArXiv*, URL https://arxiv.org/abs/2010.11929. (Cited at pages 6 e 19)

FU, H., GONG, M., WANG, C., BATMANGHELICH, K., ZHANG, K. e TAO, D. (2018), «Geometry-Consistent Generative Adversarial Networks for One-Sided Unsupervised Domain Mapping», *ArXiv*, URL https://arxiv.org/abs/1809.05852. (Cited at page 16)

GATYS, L. A., ECKER, A. S. e BETHGE, M. (2015), «A Neural Algorithm of Artistic Style», *ArXiv*, URL https://arxiv.org/abs/1508.06576. (Cited at page 9)

GLOROT, X. e BENGIO, Y. (2010), «Understanding the difficulty of training deep feedforward neural networks», *PMLR*, URL https://proceedings.mlr.press/v9/glorot10a.html. (Cited at page 29)

HASSANI, A., WALTON, S., SHAH, N., ABUDUWEILI, A., LI, J. e SHI, H. (2021), «Escaping the Big Data Paradigm with Compact Transformers», *ArXiv*, URL https://arxiv.org/abs/2104.05704. (Cited at pages 6 e 28)

IIZUKA, S. e EDGAR SIMO-SERRA, H. I. (2016), «Let there be color!: joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification», *ACM Digital Library*, URL https://dl.acm.org/doi/10.1145/2897824.2925974. (Cited at page 9)

ISOLA, P., ZHU, J.-Y., ZHOU, T. e EFROS, A. A. (2017), «Image-to-Image Translation with Conditional Adversarial Networks», *ArXiv*, URL https://arxiv.org/abs/1611.07004. (Cited at pages 8, 15 e 17)

JOHNSON, J., ALAHI, A. e FEI-FEI, L. (2016), «Perceptual Losses for Real-Time Style Transfer and Super-Resolution», *ArXiv*, URL https://arxiv.org/abs/1603.08155. (Cited at page 15)

KIM, J., KIM, M., KANG, H. e LEE, K. (2019), «U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation», *ArXiv*, URL https://arxiv.org/abs/1907.10830. (Cited at page 18)

LEDIG, C., THEIS, L., HUSZAR, F., CABALLERO, J., CUNNINGHAM, A., ACOSTA, A., AITKEN, A., TEJANI, A., TOTZ, J., WANG, Z. e SHI, W. (2017), «Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network», *ArXiv*, URL https://arxiv.org/abs/1609.04802. (Cited at page 8)

MAO, X., LI, Q., XIE, H., LAU, R. Y., WANG, Z. e SMOLLEY, S. P. (2016), «Least Squares Generative Adversarial Networks», *ArXiv*, URL https://arxiv.org/abs/1611.04076. (Cited at page 29)

PANG, Y., LIN, J., QIN, T. e CHEN, Z. (2021), «Image-to-Image Translation: Methods and Applications», *IEEE*, URL https://ieeexplore.ieee.org/document/9528943. (Cited at page 18)

RONNEBERGER, O., FISCHER, P. e BROX, T. (2015), «U-Net: Convolutional Networks for Biomedical Image Segmentation», *ArXiv*, URL https://arxiv.org/abs/1505.04597. (Cited at page 15)

SHRIVASTAVA, A., PFISTER, T., TUZEL, O., SUSSKIND, J., WANG, W. e WEBB, R. (2016), «Learning from Simulated and Unsupervised Images through Adversarial Training», *ArXiv*, URL https://arxiv.org/abs/1612.07828. (Cited at page 30)

TANG, H., LIU, H., XU, D., TORR, P. H. e SEBE, N. (2019a), «Image Generators with Conditionally-Independent Pixel Synthesis», *ArXiv*, URL https://arxiv.org/abs/1911.11897. (Cited at page 20)

TANG, H., LIU, H., XU, D., TORR, P. H. e SEBE, N. (2021), «AttentionGAN: Unpaired Image-to-Image Translation using Attention-Guided Generative Adversarial Networks», *ArXiv*, URL https://arxiv.org/abs/1911.11897. (Cited at page 18)

TANG, Y., TANG, Y., SANDFORT, V., XIAO, J. e SUMMERS., R. M. (2019b), «TUNA-Net: Task-oriented UNsupervised Adversarial Network for Disease Recognition in Cross-Domain Chest X-rays», *ArXiv*, URL https://arxiv.org/abs/1908.07926. (Cited at pages 11 e 16)

TORBUNOV, D., HUANG, Y., YU, H., HUANG, J., YOO, S., LIN, M., VIREN, B. e REN, Y. (2022), «UVCGAN: UNet Vision Transformer cycle-consistent GAN for unpaired image-to-image translation», *ArXiv*, URL https://arxiv.org/abs/2203.02557. (Cited at pages 18 e 25)

VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L. e POLOSUKHIN, I. (2017), «Attention Is All You Need», *ArXiv*, URL `https://arxiv.org/abs/1706.03762`. (Cited at pages 3 e 23)

WEI, M., SHEN, Y., WANG, Y., XIE, H., QIN, J. e WANG, F. L. (2023), «RainDiffusion: When Unsupervised Learning Meets Diffusion Models for Real-world Image Deraining», *ArXiv*, URL `https://arxiv.org/pdf/1908.07926.pdf`. (Cited at page 11)

XUE, W., ZHOU, T., WEN, Q., GAO, J., DING, B. e JIN, R. (2023), «Make Transformer Great Again for Time Series Forecasting: Channel Aligned Robust Dual Transformer», *ArXiv*, URL `https://arxiv.org/abs/2305.12095`. (Cited at page 8)

ZHAO, Y., WU, R. e DONG, H. (2020), «Unpaired Image-to-Image Translation using Adversarial Consistency Loss», *ArXiv*, URL `https://arxiv.org/abs/2003.04858`.

ZHOU, B., KHOSLA, A., LAPEDRIZA, A., OLIVA, A. e TORRALBA, A. (2016), «Learning Deep Features for Discriminative Localization», *IEEE*, URL `https://ieeexplore.ieee.org/document/7780688`. (Cited at page 18)

ZHU, J.-Y., PARK, T., ISOLA, P. e EFROS, A. A. (2017), «Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks», *ArXiv*, URL `https://arxiv.org/abs/1703.10593`. (Cited at pages 8, 10, 11, 13, 15 e 19)

Siti Web Consultati

- Medium – `www.medium.com`
- Towards Data Science – `www.towardsdatascience.com`
- ResearchGate – `www.researchgate.net`
- GeeksforGeeks – `www.geeksforgeeks.org`
- Wikipedia – `www.wikipedia.org`

Ringraziamenti

Il primo ringraziamento va ai miei genitori, che mi hanno sostenuto in tutto il percorso universitario.

Un altro grande ringraziamento va ai colleghi e amici Alessandro Marcolini e Alessandro Muscatello, oltre che alla mia ragazza Marta, con cui ho potuto scambiare idee e confrontarmi in momenti difficili.

Vorrei ringraziare il Prof. Ursino, insieme al Dott. Luca Virgili, per essere stati sempre molto disponibili nel dare consigli per la tesi, e avermi seguito nel percorso di stage.

Ringrazio Alice, per tutti i consigli forniti in illustrazioni grafiche, come le slide per le presentazioni. Un ultimo ringraziamento va a tutti i colleghi della Facoltà di Ingegneria, che hanno seguito i miei stessi corsi, per aver reso più leggero e divertente frequentare le varie lezioni.