

TRƯỜNG ĐẠI HỌC XÂY DỰNG HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN KHOA HỌC MÁY TÍNH



BÁO CÁO ĐỒ ÁN
THỊ GIÁC MÁY TÍNH

Đề tài: Ứng dụng GAN trong phục hồi vùng ảnh
bị mất (Image Infilling)

Giảng viên hướng dẫn: ThS. Thái Thị Nguyệt

Sinh viên thực hiện: Phạm Hồng Thái - 0127067 - 67CS1
Lã Minh Khánh - 4004267 - 67CS1
Trịnh Quỳnh Anh - 0279367 - 67CS1
Nguyễn Hải Cường - 0174067 - 67CS1

Hà Nội, ngày 11 tháng 05 năm 2025

MỤC LỤC

LỜI NÓI ĐẦU	4
Chương I: Giới thiệu	5
1.1 Giới thiệu đề tài	6
1.2 Giới thiệu bộ dữ liệu	7
1.3 Mục tiêu, đối tượng và phạm vi	8
Chương II: Cơ sở lý thuyết	9
2.1 Image Infilling là gì?	9
2.2 Mạng CNN	9
2.2.1 Mở đầu về mạng neuron	9
2.2.2 Mạng CNN	11
2.3 Mạng GAN	24
2.3.1 Generator dựa trên Context Encoder	26
2.3.2 Discriminator có kiến trúc CNN	30
2.3.3 Hàm mất mát trong GAN	34
2.4 Quy trình huấn luyện mô hình GAN cho Image Infilling	36
2.4.1 Chuẩn hóa ảnh đầu vào	36
2.4.2 Tạo mặt nạ M để che một vùng trung tâm của ảnh	37
2.4.3 Sinh ảnh phục hồi	38
2.4.4 Ghép lại ảnh hoàn chỉnh từ phần còn lại và phần được tạo	39
2.4.5 Mục tiêu huấn luyện	39
2.4.6 Hàm mất mát tổng thể	40
2.4.7 Quá trình huấn luyện luân phiên	40
2.5 Kỹ thuật khôi phục ảnh Image Inpainting	41
2.5.1 Khái niệm	41
2.5.2 Các kỹ thuật phổ biến	41
2.5.3 Quy trình khôi phục ảnh bằng GAN	42
2.5.4 Chiến lược mặt nạ (Masking Strategies)	42
2.5.5 Các yếu tố ảnh hưởng chất lượng phục hồi	42
2.6 Các chỉ số đánh giá chất lượng ảnh	43
2.6.1 PSNR	43
2.6.2 RMSE	44
2.6.3 MAE	45
2.6.4 SSIM	46

2.6.5 FID	48
Chương III: Thực nghiệm	51
3.1. Dữ liệu đầu vào	51
3.2. Cài đặt mô hình	52
3.2.1. Cài đặt thư viện	52
3.2.2. Thiết lập tham số mô hình	53
3.2.3. Cài đặt kiến trúc mô hình	57
3.2.4. Thiếp lập cấu hình huấn luyện mạng GAN	61
3.3. Kết quả thực nghiệm	67
LỜI BẠT	70
TÀI LIỆU THAM KHẢO	71

LỜI NÓI ĐẦU

Chúng em xin gửi lời cảm ơn sâu sắc đến cô Thái Thị Nguyệt, người đã tận tâm giảng dạy và truyền đạt những kiến thức quý báu trong suốt quá trình học tập học phần "Đồ án Thị giác máy tính". Nhờ sự hướng dẫn nhiệt tình và tận tâm của cô, chúng em đã có được nền tảng kiến thức vững chắc để bước đầu khám phá và tiếp cận sâu hơn với chuyên ngành, cũng như tự tin thực hiện các nghiên cứu và sáng tạo trong tương lai.

Chúng em cũng xin chân thành cảm ơn cô vì đã không ngừng hỗ trợ và đồng hành cùng chúng em trong suốt quá trình thực hiện đề tài này. Những lời chỉ bảo và góp ý của cô chính là nguồn động lực lớn giúp chúng em hoàn thành tốt nhiệm vụ được giao.

Bên cạnh đó, chúng em xin gửi lời tri ân đến gia đình, bạn bè và các anh chị tiền bối đã luôn động viên và cung cấp cho chúng em những tài liệu, thông tin hữu ích trong suốt quá trình thực hiện đề tài.

Dù đã nỗ lực hết mình và làm việc với tinh thần trách nhiệm cao, nhưng chắc chắn bài làm của chúng em vẫn không thể tránh khỏi những sai sót. Chúng em rất mong nhận được sự đóng góp ý kiến từ các thầy cô để bài tập được hoàn thiện hơn, đồng thời giúp chúng em tích lũy thêm kinh nghiệm quý báu cho những lần nghiên cứu sau.

Chúng em xin chân thành cảm ơn!

Ngày 11 tháng 05 năm 2024

Nhóm 13

TRỊNH QUỲNH ANH

LÃ MINH KHÁNH

PHẠM HỒNG THÁI

NGUYỄN HẢI CƯỜNG

Chương I: Giới thiệu

Nhận dạng cảnh (scene recognition) là một trong những nhiệm vụ nền tảng trong thị giác máy tính, đóng vai trò quan trọng trong việc cung cấp ngữ cảnh cho các bài toán cao cấp như nhận dạng đối tượng, phân tích hành vi, hay phục hồi nội dung ảnh. Trong khi các mô hình nhận dạng đối tượng đã đạt được nhiều thành tựu nổi bật nhờ vào bộ dữ liệu ImageNet và các kiến trúc CNN mạnh mẽ, thì nhận dạng cảnh vẫn là một thách thức.

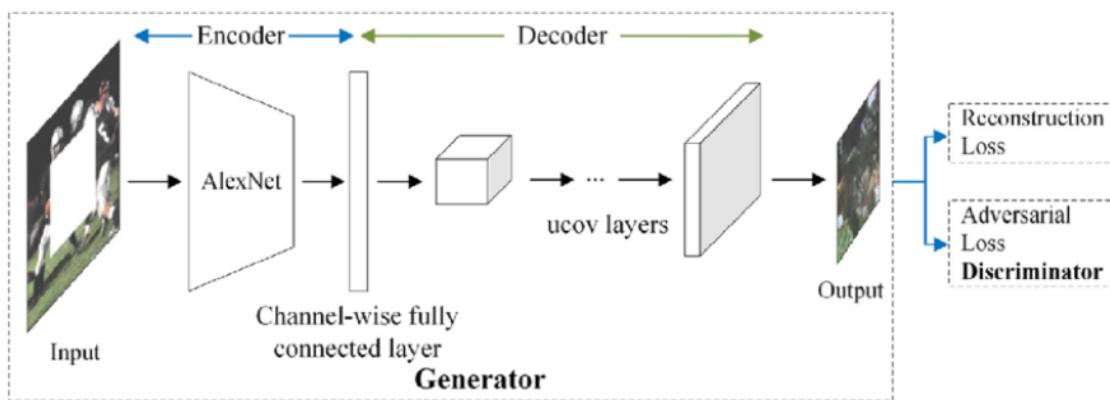
Nguyên nhân chính là do các mô hình huấn luyện trên ImageNet – vốn tập trung vào nhận diện đối tượng (object-centric) – không thể khai thác tốt các mối quan hệ không gian và đặc trưng ngữ cảnh cần thiết cho nhận diện cảnh (scene-centric). Để giải quyết vấn đề này, bộ dữ liệu **Places365** đã được xây dựng, với hơn 1.8 triệu hình ảnh và 365 loại cảnh (scene categories), giúp các mô hình học sâu hiểu sâu hơn về bối cảnh và ngữ cảnh trong ảnh.

Với những đặc điểm đa dạng, giàu ngữ cảnh và được xác thực chất lượng, Places365 không chỉ được sử dụng rộng rãi trong nhận dạng cảnh mà còn rất phù hợp cho các bài toán tái tạo ảnh, nơi việc “đoán” nội dung ảnh bị thiếu dựa vào ngữ cảnh là nhiệm vụ then chốt. Trong đề tài này, chúng ta sẽ áp dụng GAN – một mô hình sinh đối kháng – kết hợp với dữ liệu từ Places365 để giải quyết bài toán **khôi phục vùng bị che khuất trong ảnh** một cách tự nhiên và hợp lý.

1.1.Giới thiệu đề tài

Bài toán **Image Infilling** (khôi phục vùng thiếu trong ảnh) là một hướng nghiên cứu quan trọng trong xử lý ảnh, với mục tiêu tái tạo phần ảnh bị mất sao cho kết quả vừa chính xác về mặt nội dung, vừa thuyết phục về mặt thị giác. Thay vì sử dụng các phương pháp nội suy cổ điển hoặc lọc tuyến tính vốn thiếu tính tự nhiên, các mô hình học sâu, đặc biệt là **Generative Adversarial Networks (GANs)**, đã chứng minh hiệu quả vượt trội nhờ khả năng học các đặc trưng ngữ cảnh phức tạp trong ảnh.

Trong đề tài này, chúng em xây dựng một mô hình **Context Encoder GAN** (theo Pathak et al., 2016):



Trong đó:

- **Generator** có kiến trúc **encoder–decoder** tiếp nhận ảnh bị che và sinh ra phần bị mất.
- **Không sử dụng kết nối tắt (skip connections)** như trong U-Net.
- **Discriminator** là mạng CNN nhằm phân biệt ảnh thật với ảnh đã được khôi phục.
- Mô hình được huấn luyện trên **bộ dữ liệu Places365**, với ảnh đầu vào là ảnh RGB 256×256 bị che một vùng 128×128 ở trung tâm.

Ngoài **adversarial loss**, mô hình còn sử dụng L1 loss giữa vùng ảnh được sinh và vùng gốc để tăng độ chính xác chi tiết. Kết quả được đánh giá bằng các chỉ số như **PSNR**, **SSIM**, và đánh giá trực quan.

1.2.Giới thiệu bộ dữ liệu

Để nâng cao chất lượng khôi phục ảnh trong các mô hình Image Infilling, việc lựa chọn bộ dữ liệu phù hợp là rất quan trọng. Bộ dữ liệu Places, phiên bản mở rộng Places2 và phiên bản Places365 là những tập dữ liệu cảnh quan trọng, được thiết kế để huấn luyện các mô hình học sâu trong nhận dạng cảnh và phục hồi ảnh.

- **Places:** Giới thiệu tại NIPS 2014, bao gồm hơn 7 triệu ảnh được gán nhãn theo 205 danh mục cảnh khác nhau. Mỗi ảnh được thu thập từ internet và được xác thực thông qua Amazon Mechanical Turk để đảm bảo chất lượng dữ liệu.
- **Places2:** Phiên bản mở rộng với hơn 10 triệu ảnh thuộc 401 danh mục cảnh, cung cấp dữ liệu phong phú và đa dạng hơn, hỗ trợ tốt cho việc huấn luyện các mô hình học sâu phức tạp .
- **Places365:** là phiên bản tiêu chuẩn hóa của bộ dữ liệu cảnh vật (scene-centric) (là tập con của phiên bản **Places2**), được phát triển bởi MIT nhằm phục vụ các bài toán nhận dạng cảnh và thị giác máy tính.
 - **Số lượng ảnh:** hơn 1.8 triệu ảnh huấn luyện, 365 loại cảnh (indoor và outdoor).
 - **Kích thước ảnh chuẩn hóa:** 256×256 RGB.
 - **Đặc trưng:** hình ảnh chứa ngữ cảnh phong phú, phức tạp hơn nhiều so với ImageNet, rất phù hợp để huấn luyện mô hình sinh ảnh (Image Generation / Infilling).

Việc lựa chọn sử dụng Places365 thường phổ biến hơn do được dùng làm chuẩn trong nhiều nghiên cứu GAN, phân loại cảnh, image infilling; khả năng truy cập công khai, dễ tải; khả năng cân bằng lớp (được xử lý và tổ chức lại -> dễ quản lý)

1.3.Mục tiêu, đối tượng và phạm vi

Mục tiêu

- Xây dựng mô hình Image Infilling sử dụng kiến trúc Context Encoder GAN:
 - Generator là mạng **encoder–decoder không có skip connections**.
 - Discriminator là mạng CNN đánh giá tính chân thực của phần được khôi phục.
- Huấn luyện trên bộ dữ liệu Places365, ảnh che vùng trung tâm 128×128.
- Tối ưu hóa mô hình với **L1 loss + Adversarial loss**.
- Đánh giá chất lượng phục hồi ảnh bằng PSNR, SSIM và quan sát trực quan.

Đối tượng nghiên cứu

- Bài toán: Image Inpainting.
- Mô hình GAN kiểu **Context Encoder**.
- Bộ dữ liệu Places365 – 365 loại cảnh, ảnh 256×256.
- Loss: L1, Adversarial.

Phạm vi

- Dữ liệu sử dụng: MNIST
- Không sử dụng kiến trúc U-Net hay Transformer.
- Kỹ thuật che ảnh: tạo mặt nạ hình chữ nhật che vùng giữa ảnh.
- Không áp dụng attention, perceptual loss, hay các kỹ thuật khử nhiễu nâng cao.

Chương II: Cơ sở lý thuyết

2.1.Image Infilling là gì?

Image Infilling (hay còn gọi là Image Inpainting) là một bài toán phục hồi ảnh, trong đó một mô hình được yêu cầu “điền vào” những vùng bị thiếu hoặc bị che khuất trong một bức ảnh sao cho kết quả cuối cùng trông tự nhiên và liền mạch. Bài toán này đòi hỏi mô hình không chỉ học được cấu trúc cục bộ của ảnh mà còn phải hiểu ngữ cảnh tổng thể (tổng cục) để sinh ra các chi tiết hợp lý.

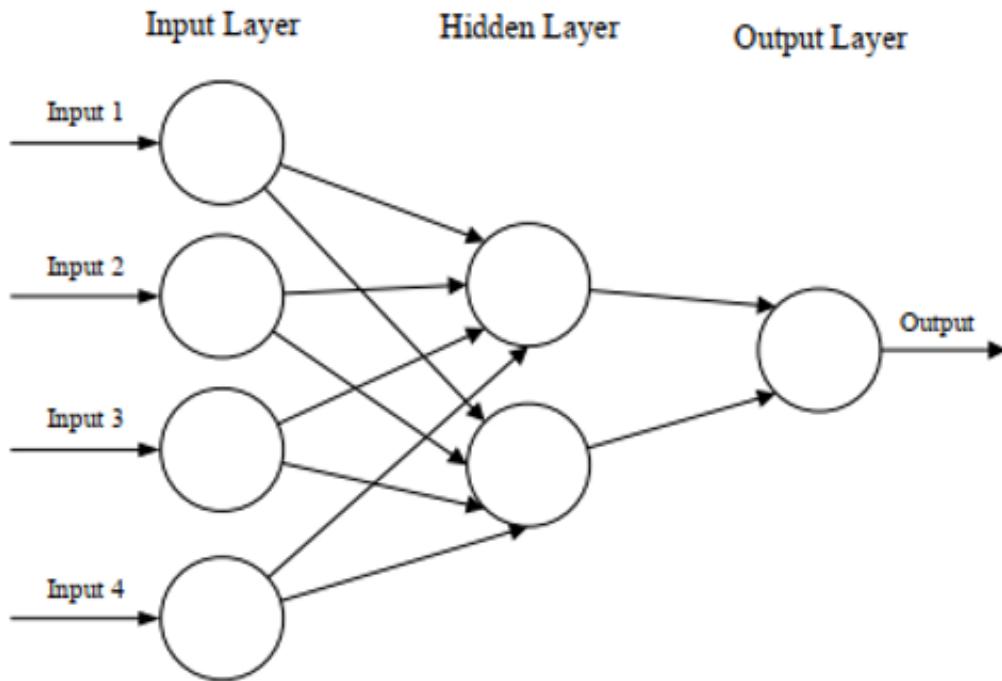
Ứng dụng thực tiễn của Image Infilling:

- Phục hồi ảnh cũ, ảnh bị hư hỏng.
- Xóa vật thể không mong muốn trong ảnh.
- Nội suy hoặc hoàn thiện ảnh từ dữ liệu cảm biến thiếu.
- Tiền xử lý dữ liệu ảnh cho các hệ thống học máy.

2.2.Mạng CNN

2.2.1 Mở đầu

Trong những năm gần đây, lĩnh vực trí tuệ nhân tạo (artificial intelligence) đang có những bước phát triển đột phá. Điều này diễn ra chủ yếu là nhờ sự phát triển của các mô hình Deep Learning hay các cấu trúc mạng neural nhân tạo. Artificial Neural Network là một nhánh mô hình học máy (Machine Learning) được lấy cảm hứng từ bộ não của con người, sử dụng các nút (còn gọi là các neuron) được liên kết với nhau trong một cấu trúc phân lớp giống như bộ não của con người.



Một cấu trúc mạng neural thường bao gồm 3 thành phần chính:

- Tầng đầu vào (Input layer)
- Tầng đầu ra (Output layer)
- Các tầng ẩn (Multiple hidden layers)

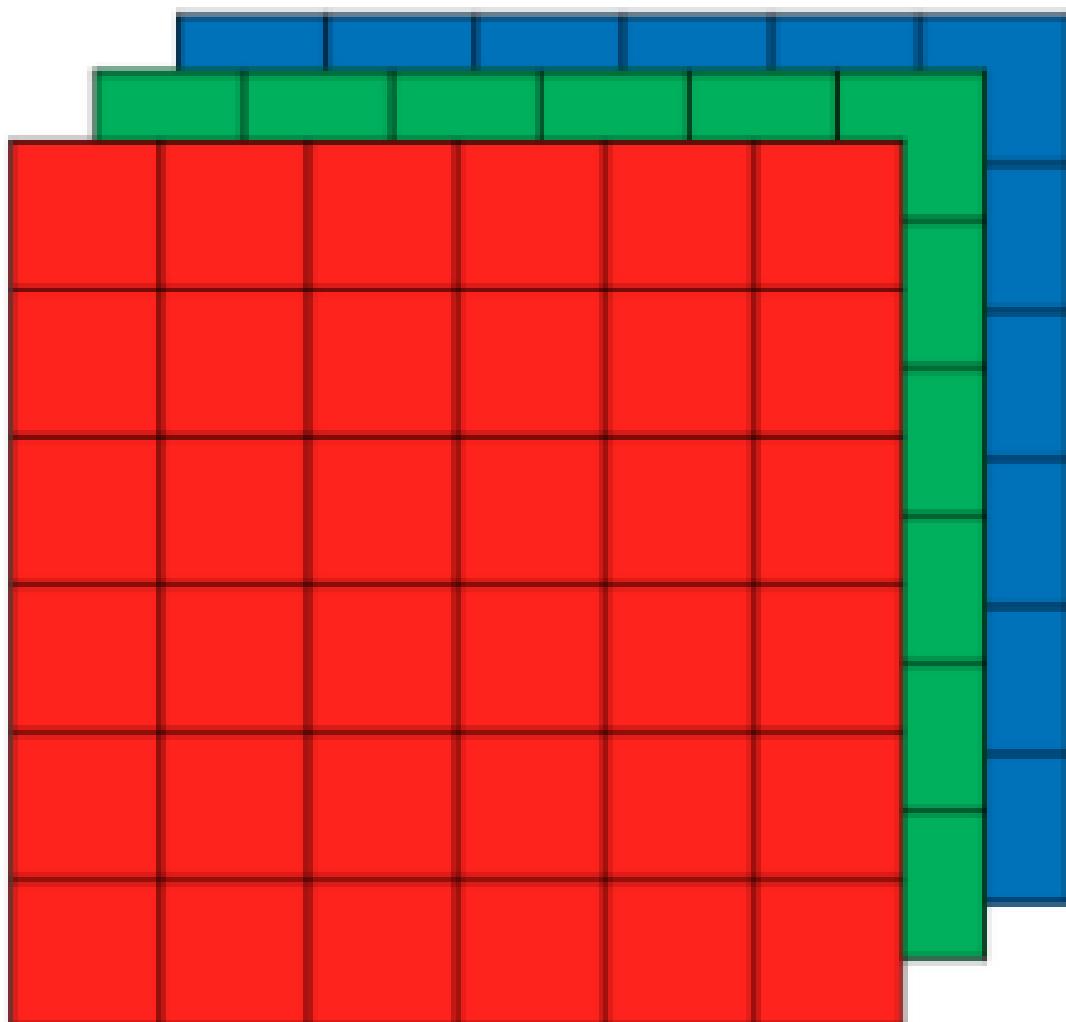
Dữ liệu đầu vào được biểu diễn dưới dạng một vector nhiều chiều đưa vào tầng đầu vào. Sau đó dữ liệu sẽ được trải qua xử lý tính toán ở các tầng ẩn (có thể có một hoặc nhiều tầng). Mỗi lớp ẩn phân tích dữ liệu đầu ra từ lớp trước, xử lý dữ liệu đó sâu hơn và rồi chuyển dữ liệu sang lớp tiếp theo. Cuối cùng dữ liệu đầu ra của tầng ẩn cuối cùng sẽ được đưa vào tầng đầu ra để tính toán đầu ra cho bài toán.

Hai hướng tiếp cận cho cách học của một mô hình mạng neural là học giám sát và học không giám sát. Học có giám sát là học thông qua các đầu vào có gán nhãn, đối với mỗi đầu vào sẽ có một tập hợp các giá trị đầu vào (vector) và một hoặc nhiều giá trị đầu ra. Mục tiêu của mô hình này là giảm bớt lỗi phân loại tổng thể, thông qua tính toán chính xác giá trị đầu ra của ví dụ thông qua quá trình huấn luyện. Học không giám sát khác ở chỗ tập huấn luyện không bao gồm bất kỳ nhãn nào. Mức độ chính xác của mô hình thường được xác định bằng việc mạng có thể giảm hoặc tăng giá trị một hàm mất mát.

Từ mô hình mạng neural đơn giản trên, có nhiều dạng neural network đã ra đời để phục vụ cho các mục đích khác nhau, với các kiến trúc khác nhau như mạng CNN (Convolutional Neural Network), mạng RNN (Recurrent Neural Network),...

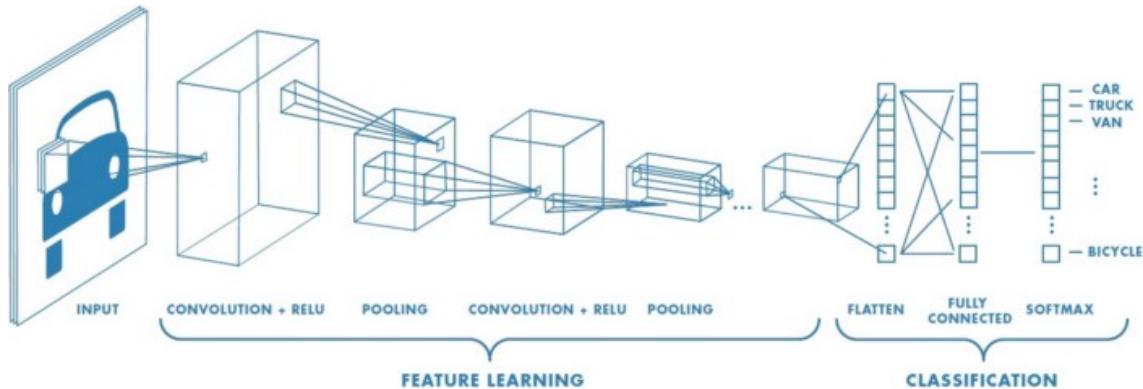
2.2.2 Mạng CNN

- Mạng nơ-ron tích chập (Convolutional Neural Network - CNN) là một loại mạng thần kinh nhân tạo chuyên xử lý dữ liệu hình ảnh. CNN bao gồm các lớp tích chập (Convolutional Layers), lớp phi tuyến (Activation Layers), lớp chuẩn hóa (Batch Normalization), và lớp gộp (Pooling Layers) giúp trích xuất đặc trưng hiệu quả.
- CNN phân loại hình ảnh bằng cách lấy 1 hình ảnh đầu vào, xử lý và phân loại nó theo các hạng mục nhất định (Ví dụ: Chó, Mèo, Hổ, ...). Máy tính coi hình ảnh đầu vào là 1 mảng pixel và nó phụ thuộc vào độ phân giải của hình ảnh. Dựa trên độ phân giải hình ảnh, máy tính sẽ thấy $H \times W \times D$ (H : Chiều cao, W : Chiều rộng, D : Độ dày). Ví dụ: Hình ảnh là mảng ma trận RGB $6 \times 6 \times 3$ (3 ở đây là giá trị RGB).



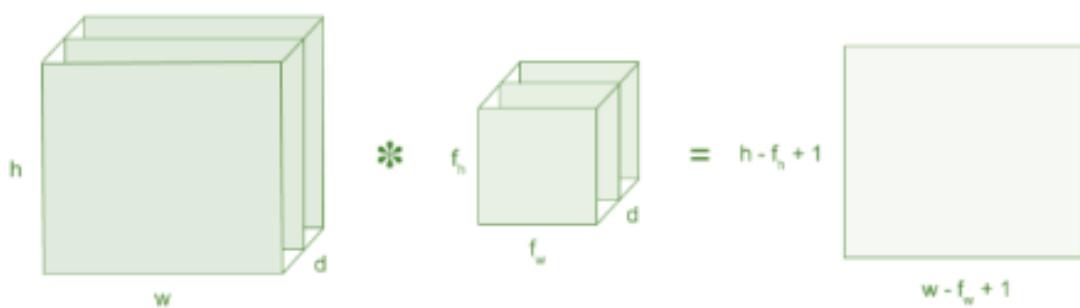
$6 \times 6 \times 3$

- Về kỹ thuật, mô hình CNN để training và kiểm tra, mỗi hình ảnh đầu vào sẽ chuyển nó qua 1 loạt các lớp tích chập với các bộ lọc (Kernels), tổng hợp lại các lớp được kết nối đầy đủ (Full Connected) và áp dụng hàm Softmax để phân loại đối tượng có giá trị xác suất giữa 0 và 1. Hình dưới đây là toàn bộ luồng CNN để xử lý hình ảnh đầu vào và phân loại các đối tượng dựa trên giá trị.



2.2.2.1 Lớp tích chập - Convolution Layer

- Tích chập là lớp đầu tiên để trích xuất các tính năng từ hình ảnh đầu vào. Tích chập duy trì mối quan hệ giữa các pixel bằng cách tìm hiểu các tính năng hình ảnh bằng cách sử dụng các ô vuông nhỏ của dữ liệu đầu vào. Nó là 1 phép toán có 2 đầu vào như ma trận hình ảnh và 1 bộ lọc hoặc hạt nhân.
- An image matrix (volume) of dimension **($h \times w \times d$)**
- A filter **($f_h \times f_w \times d$)**
- Outputs a volume dimension **($h - f_h + 1 \times (w - f_w + 1) \times 1$)**



- Xem xét 1 ma trận 5×5 có giá trị pixel là 0 và 1. Ma trận bộ lọc 3×3 như hình bên dưới.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

*

1	0	1
0	1	0
1	0	1

5 x 5 – Image Matrix

3 x 3 – Filter Matrix

- Sau đó, lớp tích chập của ma trận hình ảnh 5 x 5 nhân với ma trận bộ lọc 3 x 3 gọi là 'Feature Map' như hình bên dưới.

- Bước 1:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

4		

Image

Convolved
Feature

- Bước 2:

1	1 <small>*1</small>	1 <small>*0</small>	0 <small>*1</small>	0
0	1 <small>*0</small>	1 <small>*1</small>	1 <small>*0</small>	0
0	0 <small>*1</small>	1 <small>*0</small>	1 <small>*1</small>	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	

Convolved Feature

1	1	1 <small>*1</small>	0 <small>*0</small>	0 <small>*1</small>
0	1	1 <small>*0</small>	1 <small>*1</small>	0 <small>*0</small>
0	0	1 <small>*1</small>	1 <small>*0</small>	1 <small>*1</small>
0	0	1	1	0
0	1	1	0	0

Image

4	3	4

Convolved Feature

- Bước 4:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	4
2		

Convolved Feature

- Bước 5:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	4
2	4	

Convolved Feature

- Bước 6:

1	1	1	0	0
0	1	1 _{x1}	1 _{x0}	0 _{x1}
0	0	1 _{x0}	1 _{x1}	1 _{x0}
0	0	1 _{x1}	1 _{x0}	0 _{x1}
0	1	1	0	0

Image

4	3	4
2	4	3

Convolved Feature

1	1	1	0	0
0	1	1	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0 _{x0}	0 _{x1}	1 _{x0}	1	0
0 _{x1}	1 _{x0}	1 _{x1}	0	0

Image

4	3	4
2	4	3
2		

Convolved Feature

- Bước 8:

1	1	1	0	0
0	1	1	1	0
0	0 _{x1}	1 _{x0}	1 _{x1}	1
0	0 _{x0}	1 _{x1}	1 _{x0}	0
0	1 _{x1}	1 _{x0}	0 _{x1}	0

Image

4	3	4
2	4	3
2	3	

Convolved Feature

- Bước 9:

1	1	1	0	0
0	1	1	1	0
0	0	1 _{w1}	1 _{w0}	1 _{w1}
0	0	1 _{w0}	1 _{w1}	0 _{w0}
0	1	1 _{w1}	0 _{w0}	0 _{w1}

Image

4	3	4
2	4	3
2	3	4

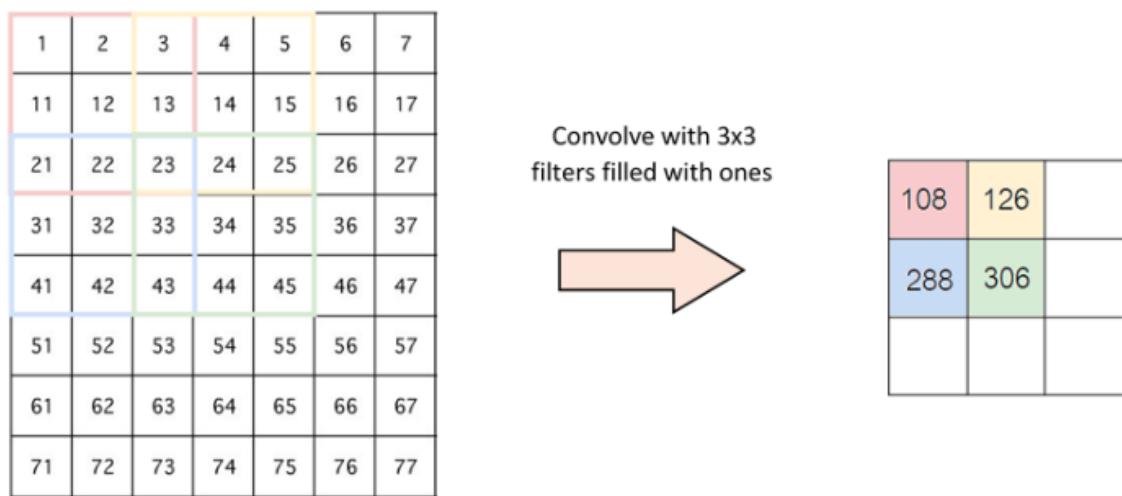
Convolved Feature

- Sự kết hợp của 1 hình ảnh với các bộ lọc khác nhau có thể thực hiện các hoạt động như phát hiện cạnh, làm mờ và làm sắc nét bằng cách áp dụng các bộ lọc. Ví dụ dưới đây cho thấy hình ảnh tích chập khác nhau sau khi áp dụng các Kernel khác nhau.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

2.2.2.2 Bước nhảy - Stride

- Stride là số pixel thay đổi trên ma trận đầu vào. Khi stride là 1 thì ta di chuyển các kernel 1 pixel. Khi stride là 2 thì ta di chuyển các kernel đi 2 pixel và tiếp tục như vậy. Hình dưới là lớp tích chập hoạt động với stride là 2.

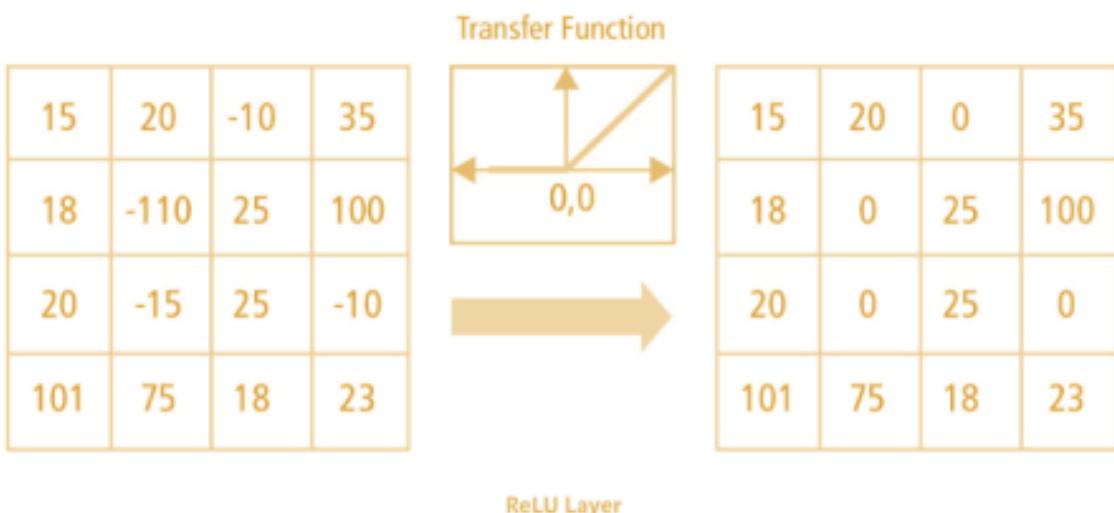


2.2.2.3 Đường viền - padding

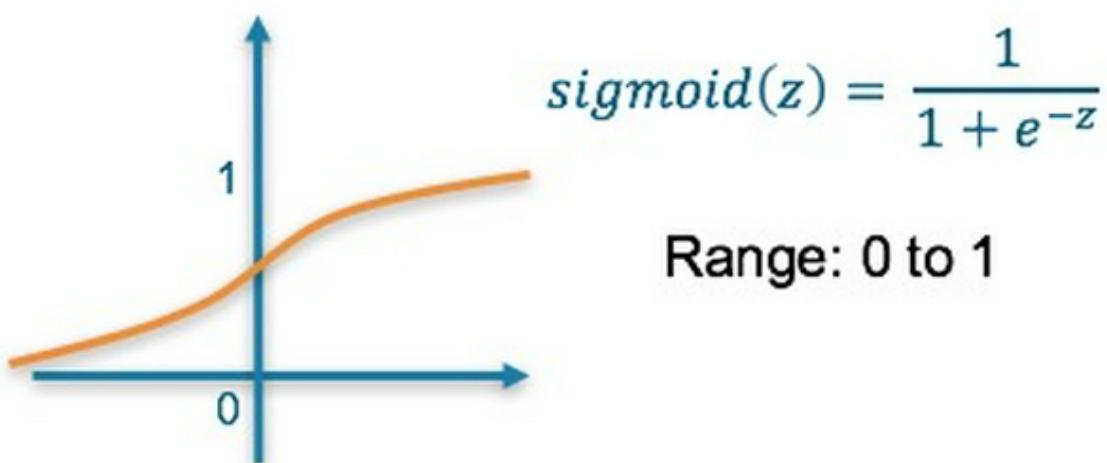
- Đôi khi kernel không phù hợp với hình ảnh đầu vào. Ta có 2 lựa chọn:
 - Chèn thêm các số 0 vào 4 đường biên của hình ảnh (padding).
 - Cắt bớt hình ảnh tại những điểm không phù hợp với kernel.

2.2.2.4 Hàm phi tuyến - RELU

- ReLU viết tắt của Rectified Linear Unit, là 1 hàm phi tuyến. Với đầu ra là: $f(x) = \max(0, x)$.
- Tại sao ReLU lại quan trọng: ReLU giới thiệu tính phi tuyến trong ConvNet. Vì dữ liệu trong thế giới mà chúng ta tìm hiểu là các giá trị tuyến tính không âm.



- Có 1 số hàm phi tuyến khác như **tanh**, **sigmoid** cũng có thể được sử dụng thay cho ReLU. Hầu hết người ta thường dùng ReLU vì nó có hiệu suất tốt.

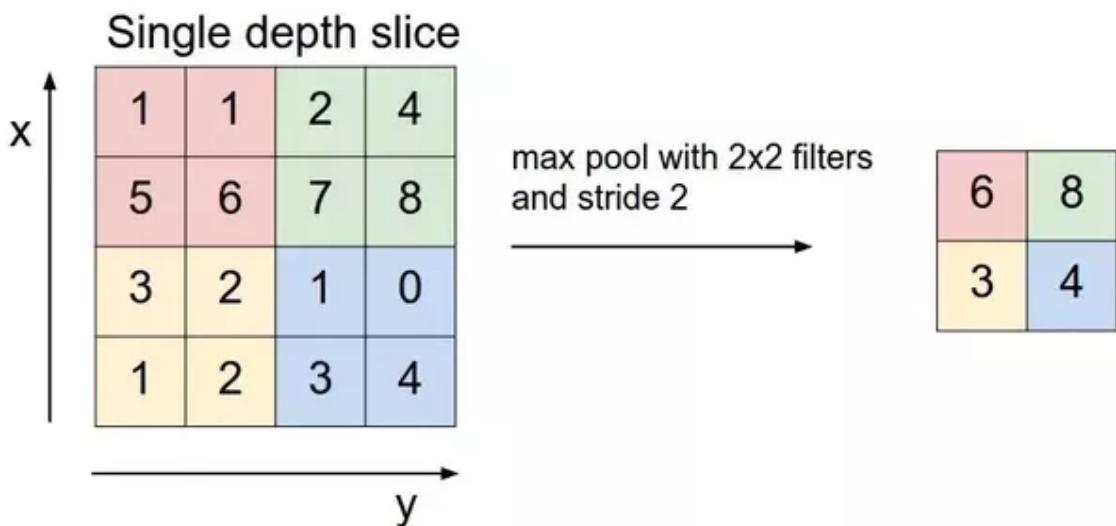


Tanh

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

2.2.2.5 Lớp gộp - Pooling Layer

- Lớp pooling sẽ giảm bớt số lượng tham số khi hình ảnh quá lớn. Không gian pooling còn được gọi là lấy mẫu con hoặc lấy mẫu xuống làm giảm kích thước của mỗi map nhưng vẫn giữ lại thông tin quan trọng. Các pooling có thể có nhiều loại khác nhau:
 - Max Pooling
 - Average Pooling
 - Sum Pooling
- Max pooling lấy phần tử lớn nhất từ ma trận đối tượng, hoặc lấy tổng trung bình. Tổng tất cả các phần tử trong map gọi là sum pooling.



2.2.2.6 Tóm tắt

- Đầu vào của lớp tích chập là hình ảnh
- Chọn đối số, áp dụng các bộ lọc với các bước nhảy, padding nếu cần. Thực hiện tích chập cho hình ảnh và áp dụng hàm kích hoạt ReLU cho ma trận hình ảnh.
- Thực hiện Pooling để giảm kích thước cho hình ảnh.
- Thêm nhiều lớp tích chập sao cho phù hợp.
- Xây dựng đầu ra và dữ liệu đầu vào thành 1 lớp được kết nối đầy đủ (Full Connected)
- Sử dụng hàm kích hoạt để tìm đối số phù hợp và phân loại hình ảnh.

2.3.Mạng GAN

Có lẽ khi nghe đến **trí tuệ nhân tạo**, bạn sẽ liên tưởng đến các bộ phim siêu anh hùng, nơi khả năng kỳ diệu được tạo ra, hay những chú robot có khả năng nghe, nói và thể hiện cảm xúc. Tất cả những điều đó đem lại cho trí tuệ nhân tạo cụm từ mang tên **sự sáng tạo**. Vậy trong thực tế, với các kỹ thuật Deep Learning hiện nay, chúng có thể đem lại sự sáng tạo cho trí tuệ nhân tạo? Điều này đang có những bước tiến ban đầu. Và một trong những kỹ thuật giúp tạo ra sự sáng tạo cho AI đó là mạng **Generative Adversarial Networks** viết tắt là mạng GAN.

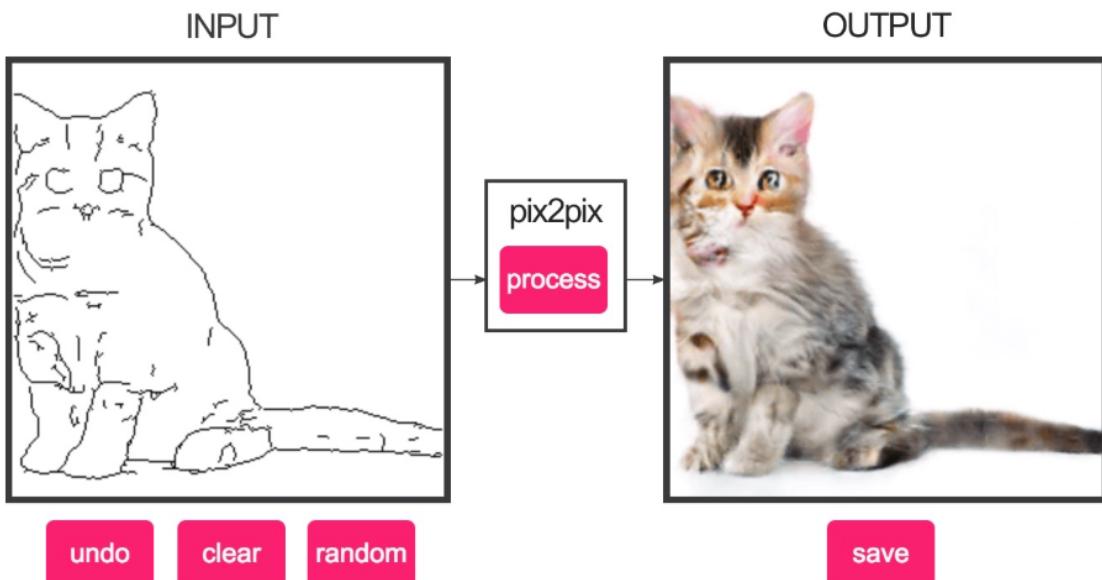
Mạng GAN được sử dụng trong việc tạo ra rất nhiều sản phẩm hay, ví dụ việc làm cho máy tính tự động vẽ một bức chân dung, hoặc thậm chí làm cho máy tính có thể tự động sáng tác ra một bản nhạc theo một phong cách nào đó - điều mà bạn nghĩ chỉ có con người mới có thể làm được. So với các cấu trúc của mạng nơ-ron khác, thì mạng GAN có thể làm được điều tưởng chừng như không thể mà các mạng nơ-ron khác không làm được.

Việc xác định một bức tranh có phải theo phong cách của một họa sĩ nào đó dễ dàng hơn nhiều so với việc tự vẽ ra bức tranh đó. Và GAN bước đầu đang đưa ta dần đến giấc mơ trí tuệ nhân tạo.

Và gần đây, có rất nhiều bài báo và sản phẩm được tạo ra từ mạng GAN, ví dụ như ứng dụng trong phát triển trò chơi. Công việc vẽ các nhân vật hoạt hình tồn rất nhiều chi phí của các nhà phát triển, do họ phải thuê họa sĩ vẽ các nhân vật hoạt hình này ở các tư thế khác nhau. Và công việc này là một công việc lặp đi lặp lại. Người ta đã sử dụng tư tưởng của mạng GAN để có thể làm cho máy tính tự động vẽ ra nhân vật hoạt hình một cách tương tự, điều đó làm cho nhà phát triển giảm được chi phí thuê nhân công. Do đó, chúng ta có thể tập trung vào các khía cạnh sáng tạo khác thay vì phải tập trung vào các công việc lặp đi lặp lại.

GAN làm gì trong thực tế?

Trọng tâm chính của mạng GAN thường được áp dụng với hình ảnh, tuy nhiên với các miền khác như âm nhạc cũng đã được triển khai trong thực tế. Và thậm chí phạm vi ứng dụng của GAN trong đời sống còn rộng hơn nhiều. Giống như trong ứng dụng dưới đây, mạng GAN có thể giúp bạn tự động tô màu một bản vẽ chằng hạn:



Mạng Pix2Pix sử dụng ý tưởng của GAN giúp tự động tô màu cho một bản vẽ thô!

GAN viết tắt cho Generative Adversarial Networks, Generative là tính từ nghĩa là khả năng sinh ra, Network có nghĩa là mạng (mô hình), còn Adversarial là đối nghịch. GAN thuộc nhóm generative model, model nghĩa là mô hình. Vậy hiểu đơn giản generative model nghĩa là mô hình có khả năng sinh ra dữ liệu. Hay nói cách khác, GAN là mô hình có khả năng sinh ra dữ liệu mới. Ví dụ như những ảnh mặt người ở dưới bạn thấy là do GAN sinh ra, không phải mặt người thật. Dữ liệu sinh ra nhìn như thật nhưng không phải thật.



Vậy cấu trúc thực sự của mạng GAN bao gồm những gì?

GAN là một trong những phương pháp học sâu hiệu quả nhất để sinh dữ liệu, được giới thiệu bởi Ian Goodfellow vào năm 2014. Một mô hình GAN bao gồm hai thành phần học đối kháng:

- **Generator (G)**: cỗ gắng tái tạo phần ảnh bị che một cách tự nhiên.
- **Discriminator (D)**: cỗ gắng phân biệt giữa ảnh thật và ảnh sinh ra bởi Generator.

Hai mạng được huấn luyện song song trong một “trò chơi hai người” (min-max game): G cố gắng đánh lừa D, còn D cố gắng không bị lừa.

Trong bài toán **Image Infilling**, đầu vào là ảnh bị che một vùng ở trung tâm, mô hình cần tái tạo phần bị che dựa vào ngữ cảnh xung quanh sao cho thuyết phục.

Trước tiên, chúng ta sẽ tìm hiểu cấu trúc thứ nhất của mạng GAN: **Generator (G)**.

2.3.1. Generator

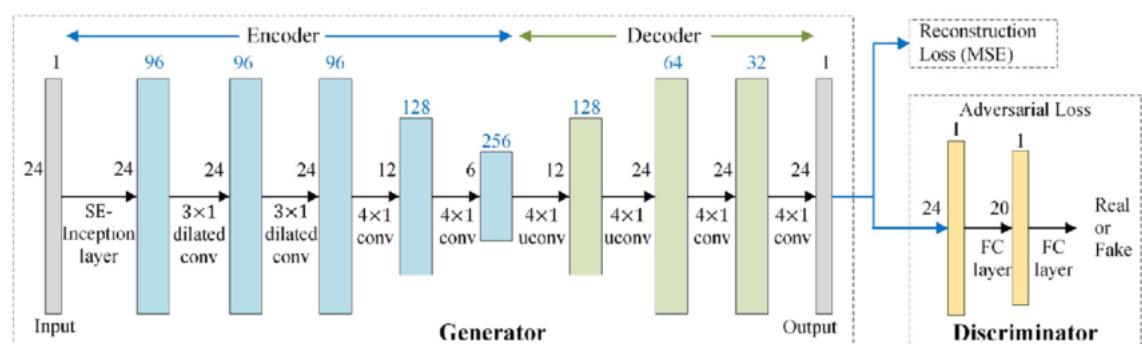
Generator – Mạng sinh ảnh bị khuyết, generator trong mô hình của chúng em dựa theo kiến trúc Context Encoder được đề xuất bởi Pathak et al. (CVPR 2016).

2.3.1.1. Vai trò

Generator có nhiệm vụ sinh ra phần ảnh bị mất sao cho liền mạch với vùng ảnh còn lại, tức là **dự đoán hợp lý nội dung vùng che**.

2.3.1.2. Kiến trúc

Trong đề tài này, Generator có cấu trúc **Encoder–Decoder** theo kiểu **Context Encoder** (Pathak et al., 2016):



- **Encoder**: gồm nhiều lớp convolution giảm dần kích thước, trích xuất đặc trưng

của ảnh đầu vào bị che.

- **Decoder:** gồm nhiều lớp transposed convolution (deconvolution) để tái dựng lại vùng bị che.

Lưu ý:

- Không có kết nối tắt (skip-connections) giữa encoder và decoder như trong U-Net.
- Generator chỉ được huấn luyện để tái tạo vùng ảnh bị che (masked region), không tái tạo toàn bộ ảnh.
- Kiến trúc này giúp mô hình học các đặc trưng ngữ cảnh quan trọng và tạo ra các chi tiết hợp lý cho vùng bị che.

2.3.1.3.Công thức tính loss cho Generator

Tổng hàm mất mát của Generator bao gồm:

$$\mathcal{L}_G = \lambda \cdot \mathcal{L}_{L1}(x, \hat{x}) + \mathcal{L}_{adv}$$

Đây là một công thức tổng hợp, kết hợp hai thành phần mất mát chính.

Trong đó:

- x : ảnh gốc (ground truth).
- $\hat{x} = G(x_{\text{masked}})$: vùng được Generator phục hồi từ ảnh bị che.
- \mathcal{L}_{L1} : sai số tuyệt đối giữa vùng sinh và vùng gốc:

$$\mathcal{L}_{L1} = \frac{1}{N} \sum_{i=1}^N |x_i - \hat{x}_i|$$

Đây là công thức của Mean Absolute Error (MAE), hay còn gọi là L1 loss. Nó đo lường sự khác biệt trung bình về cường độ pixel giữa ảnh gốc (x) và ảnh được tạo bởi Generator (\hat{x})

- \mathcal{L}_{adv} : hàm mất mát đối kháng (adversarial loss):

$$\mathcal{L}_{adv} = -\log(D(\hat{x}))$$

Đây là một dạng phổ biến của adversarial loss cho Generator, thường được sử dụng trong GAN. Mục tiêu của Generator là giảm thiểu giá trị này, tức là làm cho Discriminator (D) có xu hướng phân loại ảnh tạo ra (\hat{x}) là ảnh thật (giá trị $D(\hat{x})$ tiến đến 1). Giá trị $-\log(p)$ sẽ tiến đến 0 khi p tiến đến 1.

- λ : hệ số cân bằng giữa độ chính xác nội dung và độ chân thực thị giác (trong code: $\lambda = 100$).

Đây là một hyperparameter (siêu tham số) quan trọng để điều chỉnh mức độ ưu tiên giữa việc ảnh tạo ra giống với ảnh gốc về mặt nội dung (được đo bởi L_1) và việc ảnh tạo ra trông chân thực, khó phân biệt với ảnh thật (được thúc đẩy bởi L_{adv}).

Tất cả các công thức nêu trên được xây dựng theo chuẩn kiến trúc Context Encoder: Kiến trúc Context Encoder được thiết kế để học cách điền vào các vùng bị thiếu trong ảnh (inpainting) bằng cách sử dụng ngữ cảnh xung quanh. Nó kết hợp một mô hình autoencoder (là Generator) với một Discriminator, tạo thành một GAN.

- L_1 (Reconstruction Loss / Pixel-wise Loss):
 - Trong Context Encoder, mục tiêu chính của Generator là tái tạo chính xác các pixel trong vùng bị che (masked region). L_1 (Mean Absolute Error) là một lựa chọn tự nhiên và hiệu quả để đo lường sự khác biệt trực tiếp giữa các pixel được tạo ra (\hat{x}) và các pixel gốc (x) trong vùng bị che.
 - Loss này đảm bảo rằng nội dung được điền vào là chính xác về mặt pixel.

- L_{adv} (Adversarial Loss):

- Vấn đề của việc chỉ sử dụng L_1 (hoặc L_2) là ảnh được tạo ra có thể bị mờ. Để khắc phục điều này, Context Encoder giới thiệu thành phần adversarial loss.
- $L_{adv} = -\log(D(\hat{x}))$ đóng vai trò quan trọng trong việc khuyến khích Generator tạo ra các vùng ảnh trông **chân thực và sắc nét**, khó phân biệt với ảnh thật bởi Discriminator. Discriminator được huấn luyện để phân biệt vùng ảnh được tạo ra với vùng ảnh gốc, và Generator cố gắng "đánh lừa" Discriminator.

- λ (Weight for Reconstruction Loss):

- Việc sử dụng λ để cân bằng giữa L_1 và L_{adv} là một chiến lược quan trọng của Context Encoder.
- L_1 giúp đảm bảo tính **chính xác nội dung** (content accuracy), tức là các pixel được điền vào khớp với thực tế.
- L_{adv} giúp đảm bảo tính **chân thực thị giác** (perceptual realism), tức là vùng điền vào trông tự nhiên và liền mạch với ngữ cảnh xung quanh.
- Tác giả của Context Encoder đã thử nghiệm và thấy rằng việc kết hợp cả hai loại loss này mang lại kết quả tốt nhất, với λ là siêu tham số để điều chỉnh tầm quan trọng tương đối của mỗi loss. Giá trị $\lambda = 100$ được đề xuất trong bài báo gốc hoặc các triển khai phổ biến cho thấy sự ưu tiên mạnh mẽ hơn cho việc phục hồi nội dung chính xác.

Tiếp theo, chúng ta sẽ tìm hiểu cấu trúc thứ hai của mạng GAN: **Discriminator (D)**

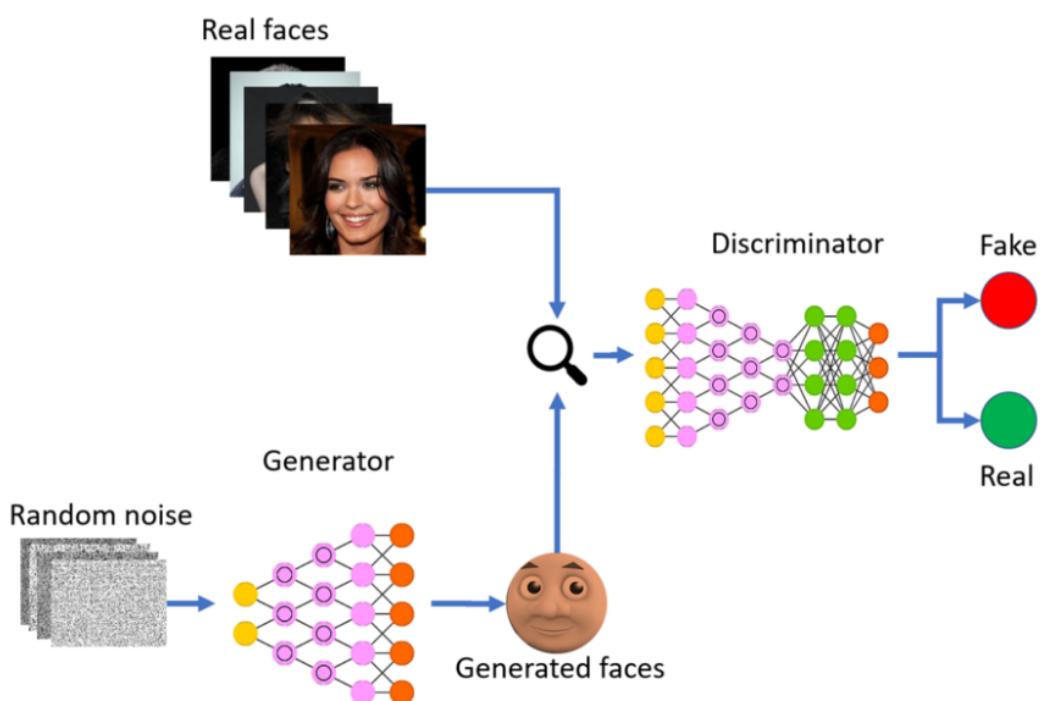
2.3.2.Discriminator

Discriminator-Mạng phân biệt thật/giả, discriminator là một mạng phân loại đơn giản gồm các tầng tích chập (Convolutional layers), nhằm phân biệt ảnh thật (gốc) và ảnh giả (ảnh đã được phục hồi bởi Generator).

2.3.2.1.Vai trò

Discriminator nhận vào một ảnh hoàn chỉnh (ảnh gốc hoặc ảnh được chắp ghép từ ảnh gốc + vùng được Generator phục hồi), và cố gắng xác định xem đây là ảnh “thật” hay ảnh “giả”.

2.3.2.2.Kiến trúc



Leaky ReLU Activation Function



Discriminator có kiến trúc CNN đơn giản, gồm:

- Nhiều lớp **convolution + batch normalization + LeakyReLU.**
- Cuối cùng là một lớp sigmoid để cho ra xác suất ảnh “thật”, phân biệt ảnh thật (gán nhãn 1) và ảnh giả (gán nhãn 0).

So sánh **Sigmoid** với **LeakyReLU**:

- đều là hàm kích hoạt (activation function) được dùng phổ biến trong mạng nơ-ron nhân tạo, nhưng chúng khác nhau rất nhiều về bản chất và ứng dụng

- **Sigmoid:**

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

- Đầu ra nằm trong khoảng (0, 1).

- Đường cong hình chữ S.

=> Sigmoid phù hợp để phân loại nhị phân ở đầu ra.

- **LeakyReLU:**

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{nếu } x > 0 \\ \alpha x & \text{nếu } x \leq 0 \end{cases}$$

- Trong đó α là một hằng số nhỏ, thường là 0.01.
- Cho phép giá trị âm “rò rỉ” thay vì bị cắt hẳn như ReLU.
- Đầu ra nằm trong khoảng

$$(-\infty, \infty)$$

LeakyReLU dùng trong các tầng ẩn:

- Trong các tầng ẩn (convolutional + batch normalization), LeakyReLU được dùng làm hàm kích hoạt phi tuyến tính.
- Mục đích:
 - Giữ cho mạng học được các đặc trưng phức tạp
 - Tránh hiện tượng "dead neuron" như ở ReLU.
 - Tránh mất gradient như ở sigmoid/tanh.

=>LeakyReLU giúp mạng học hiệu quả hơn ở các tầng ẩn.

Trong mô hình Context Encoder , discriminator được huấn luyện chỉ để đánh giá vùng bị khôi phục, chứ không đánh giá toàn ảnh.

2.3.2.3. Hàm mất mát cho Discriminator

Hàm mất mát cho Discriminator gồm 2 phần:

$$\mathcal{L}_D = -\log(D(\mathbf{x})) - \log(1 - D(\hat{\mathbf{x}}))$$

Trong đó:

- \mathbf{x} : ảnh thật (gốc).
- $\hat{\mathbf{x}}$: ảnh sinh bởi Generator.
- $D(\cdot)$: xác suất mà Discriminator cho rằng ảnh là thật.

=> Discriminator cố gắng tối đa hóa khả năng phân biệt này, còn Generator cố gắng đánh lừa Discriminator.

Công thức này là dạng hàm mất mát **Binary Cross-Entropy (BCE) Loss cho Discriminator**. Đây là hàm mất mát chuẩn mực được sử dụng trong các kiến trúc GAN gốc và nhiều biến thể của nó, bao gồm cả Context Encoder (vì Context Encoder là một dạng GAN).

- Phần đầu tiên ($-\log(D(x))$) khuyến khích Discriminator gán xác suất cao cho ảnh thật (x), tức là $D(x) \approx 1$.
- Phần thứ hai ($-\log(1 - D(\hat{x}))$) khuyến khích Discriminator gán xác suất thấp cho ảnh giả (\hat{x}), tức là $D(\hat{x}) \approx 0$.

Discriminator được huấn luyện để **tối đa hóa** biểu thức này (hoặc tối thiểu hóa nếu nó được biểu diễn với dấu cộng và không có dấu trừ phía trước, tức là coi như một hàm BCE thông thường mà Discriminator cố gắng tối thiểu hóa cho cả hai trường hợp đúng). Trong ngữ cảnh của GAN, nó thường được viết để Discriminator tối đa hóa tổng của $\log D(x)$ và $\log(1 - D(G(z)))$. Biểu thức bạn cung cấp là dạng âm của nó, nghĩa là Discriminator sẽ cố gắng **tối thiểu hóa** biểu thức này. Đây là cách viết phổ biến để thể hiện mục tiêu tối thiểu hóa khi huấn luyện bằng gradient descent.

2.3.3.Hàm mất mát trong GAN

Context Encoder huấn luyện Generator và Discriminator một cách luân phiên và độc lập bằng cách tối ưu hóa các hàm mất mát riêng của từng thành phần, dựa trên trò chơi đối kháng (adversarial game) của GAN gốc.

Tổng quát hơn, mục tiêu của GAN (bao gồm cả Context Encoder) có thể được viết dưới dạng một trò minimax:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Trong đó:

- G là Generator.
- D là Discriminator.
- $p_{data}(x)$ là phân phối dữ liệu thật.
- $p_z(z)$ là phân phối nhiễu đầu vào cho Generator.
- $D(x)$ là xác suất Discriminator dự đoán x là thật.
- $D(G(z))$ là xác suất Discriminator dự đoán ảnh giả $G(z)$ là thật.

Cách hiểu mục tiêu này trong Context Encoder:

1. Discriminator (D) cố gắng tối đa hóa $V(D, G)$:

- Nó muốn $D(x)$ (ảnh thật) gần 1, để $\log D(x)$ gần 0.
- Nó muốn $D(G(z))$ (ảnh giả) gần 0, để $\log(1 - D(G(z)))$ gần 0.
- Điều này tương ứng với hàm mất mát L_D bạn đã cung cấp: $L_D = -\mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] - \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ (chỉ cần bỏ dấu \mathbb{E} và xem xét trên một batch dữ liệu).
Tức là D sẽ tối thiểu hóa L_D này.

2. Generator (G) cỗ gắng tối thiểu hóa $V(D, G)$:

- Generator muốn $D(G(z))$ (ảnh giả) gần 1, để $1 - D(G(z))$ gần 0, và $\log(1 - D(G(z)))$ tiến đến $-\infty$.
- Tuy nhiên, như đã thảo luận trước đó, việc tối thiểu hóa $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ (hàm mất mát minimax gốc cho G) dễ bị vanishing gradient.
- Do đó, Context Encoder (và nhiều GAN hiện đại khác) sử dụng **non-saturating loss** cho Generator: $L_{adv} = -\mathbb{E}_{z \sim p_z(z)}[\log D(G(z))]$. Generator cố gắng tối thiểu hóa L_{adv} này, tức là làm cho $D(G(z))$ gần 1.

Sự kết hợp trong Context Encoder:

Hàm mất mát tổng thể của Generator trong Context Encoder không chỉ là L_{adv} mà còn bao gồm thành phần **reconstruction loss** (L_{L1}).

$$L_G = \lambda \cdot L_{L1}(\mathbf{x}, \hat{\mathbf{x}}) + L_{adv}$$

Trong đó:

- L_{L1} là Mean Absolute Error (hoặc L2 loss trong một số biến thể) giữa vùng ảnh gốc và vùng ảnh được tạo ra.
- L_{adv} là adversarial loss cho Generator ($-\log(D(\hat{x}))$).
- λ là hệ số cân bằng.

Tóm lại:

Mặc dù có công thức minimax chung cho GAN thể hiện mục tiêu đối kháng tổng thể, trong quá trình huấn luyện thực tế của Context Encoder (và hầu hết các GAN), chúng ta tối ưu hóa Discriminator và Generator bằng các hàm mất mát riêng biệt của chúng một cách luân phiên:

- Discriminator tối thiểu hóa: $L_D = -\log(\mathbf{D}(\mathbf{x})) - \log(1 - \mathbf{D}(\hat{\mathbf{x}}))$
- Generator tối thiểu hóa: $L_G = \lambda \cdot L_{L1}(\mathbf{x}, \hat{\mathbf{x}}) + (-\log(\mathbf{D}(\hat{\mathbf{x}})))$

Đây là cách tiếp cận chuẩn mực và đã được chứng minh hiệu quả cho Context Encoder, giúp nó không chỉ tạo ra các vùng ảnh chân thực mà còn chính xác về mặt nội dung.

2.4.Quy trình huấn luyện mô hình GAN cho Image Infilling

Để huấn luyện một mô hình GAN (Generative Adversarial Network) cho bài toán Image Infilling (điền vào vùng ảnh bị thiếu) sử dụng kiến trúc Context Encoder, chúng ta tuân theo các bước sau:

2.4.1.Chuẩn hóa ảnh đầu vào

Trước khi đưa ảnh vào mô hình, cần thực hiện chuẩn hóa dữ liệu. Điều này thường bao gồm:

- **Thay đổi kích thước:** Đảm bảo tất cả các ảnh có cùng kích thước (ví dụ: 256×256 pixel) để phù hợp với kiến trúc mạng.
- **Chuẩn hóa giá trị pixel:** Chuyển đổi giá trị pixel từ phạm vi $[0, 255]$ (đối với ảnh 8-bit) sang phạm vi $[-1, 1]$ hoặc $[0, 1]$. Việc này giúp mạng học ổn định hơn và tránh các vấn đề về gradient. Công thức phổ biến là $x_{norm} = (x/127.5) - 1$ để đưa về $[-1, 1]$.

Hoặc ta có cách chuẩn hóa khác:

Đầu vào ảnh được chuẩn hóa về khoảng $[-1, 1]$ hoặc $[0, 1]$ tùy theo cách xây dựng Generator:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Trong đó:

- x : giá trị pixel gốc.
- μ, σ : trung bình và độ lệch chuẩn chuẩn hóa (thường từ ImageNet: $\mu = [0.5, 0.5, 0.5]$, $\sigma = [0.5, 0.5, 0.5]$) nếu dùng tanh.

2.4.2.Tạo mặt nạ M để che một vùng trung tâm của ảnh

Bước đầu tiên là tạo ra ảnh đầu vào cho Generator. Ảnh này là phiên bản bị che một phần của ảnh gốc.

$$x_{masked} = x \cdot (1 - M) + M \cdot c$$

Trong đó:

- M : mặt nạ nhị phân, 1 ở vùng bị che. Mặt nạ M là một ma trận có cùng kích thước với ảnh x , với các giá trị 1 tại vị trí của vùng ảnh muốn che và 0 ở các vị trí còn lại.
- c : giá trị điền vào vùng che (0 hoặc 1). Đây là giá trị được dùng để "điền" tạm thời vào vùng bị che. Có thể là một giá trị cố định (ví dụ: 0 cho màu đen, 1 cho màu trắng) hoặc giá trị trung bình của ảnh. Mục đích của việc này là để Generator có một đầu vào "khung" ban đầu để bắt đầu quá trình tạo ảnh.
- x : ảnh gốc. Đây là ảnh mà chúng ta muốn điền vào vùng trống.

Công thức này tạo ra ảnh x_{masked} bằng cách giữ nguyên các pixel không bị che ($x \cdot (1 - M)$) và thay thế các pixel bị che bằng giá trị c ($M \cdot c$). Ví dụ, nếu M có giá trị 1 tại vị trí (i, j) (nghĩa là pixel đó bị che), thì $(1 - M)$ tại (i, j) sẽ là 0, và $M \cdot c$ sẽ là c tại (i, j) . Ngược lại, nếu M có giá trị 0 tại (i, j) (nghĩa là pixel đó không bị che), thì $(1 - M)$ tại (i, j) sẽ là 1, và $M \cdot c$ sẽ là 0 tại (i, j) . Kết quả là x_{masked} sẽ có vùng bị che được điền bằng c và các vùng còn lại giống ảnh gốc.

2.4.3.Sinh ảnh phục hồi

Ảnh x_{masked} sau đó được đưa vào Generator (G) để tạo ra ảnh phục hồi, đây là ảnh mà Generator "dự đoán" nội dung của vùng bị che.

$$\hat{x} = G(x_{masked})$$

Trong đó:

- \hat{x} : Ảnh được tạo bởi Generator. Đây là đầu ra của mạng Generator, đại diện cho ảnh gốc với vùng bị che đã được điền đầy.
- G : Mạng Generator. Đây là một mạng học sâu (thường là một mạng Convolutional Neural Network - CNN) có nhiệm vụ học cách tạo ra các pixel hợp lý để điền vào vùng bị che.
- x_{masked} : Ảnh gốc đã được tạo mặt nạ.

Generator nhận ảnh x_{masked} làm đầu vào, xử lý thông tin từ các vùng không bị che và sử dụng kiến thức đã học để "nội suy" và tạo ra các pixel mới cho vùng bị che. Mục tiêu của Generator là tạo ra một \hat{x} sao cho nó trông thật và khớp với ngữ cảnh của ảnh gốc.

2.4.4. Ghép lại ảnh hoàn chỉnh từ phần còn lại và phần được tạo

Bước này thường không được thể hiện rõ ràng trong công thức Loss của Context Encoder mà nằm trong cách sử dụng đầu ra của Generator. Tuy nhiên, về mặt logic, để có một ảnh hoàn chỉnh cuối cùng, chúng ta có thể kết hợp phần không bị che của ảnh gốc với phần được tạo bởi Generator:

$$x_{completed} = x \cdot (1 - M) + \hat{x} \cdot M$$

Trong đó:

- $x_{completed}$: Ảnh hoàn chỉnh sau khi điền.
- $x \cdot (1 - M)$: Phần không bị che của ảnh gốc.
- $\hat{x} \cdot M$: Phần được tạo bởi Generator, chỉ lấy vùng đã được điền.

Mặc dù Generator đã tạo ra toàn bộ ảnh \hat{x} , trong nhiều trường hợp, để đảm bảo chất lượng cao nhất cho các vùng không bị che, người ta thường lấy trực tiếp các pixel từ ảnh gốc cho những vùng này, và chỉ sử dụng các pixel được tạo bởi Generator cho vùng bị che. Điều này giúp tránh các lỗi nhỏ hoặc sự khác biệt về phong cách mà Generator có thể tạo ra ở các vùng không bị che.

2.4.5. Mục tiêu huấn luyện

Mục tiêu chính của quy trình huấn luyện là tối ưu hóa cả Generator (G) và Discriminator (D) để đạt được hai mục tiêu chính:

- **Tính chính xác nội dung (Content Accuracy):** Đảm bảo rằng nội dung được điền vào vùng bị che khớp với ảnh gốc. Điều này được thúc đẩy bởi **Reconstruction Loss (L_{L1})**.
- **Tính chân thực thị giác (Perceptual Realism):** Đảm bảo rằng ảnh được tạo ra trông chân thực và khó phân biệt với ảnh thật. Điều này được thúc đẩy bởi **Adversarial Loss (L_{adv})**.

2.4.6. Hàm mất mát tổng thể

Discriminator (D) cỗ gắng tối đa hóa $V(D, G)$ (hoặc tối thiểu hóa L_D): Mục tiêu của Discriminator là phân biệt được ảnh thật với ảnh giả (do Generator tạo ra).

$$L_D = -\log(D(x)) - \log(1 - D(\hat{x}))$$

- $-\log(D(x))$: Khuyến khích Discriminator gán xác suất cao cho ảnh thật (x), tức là $D(x) \approx 1$.
- $-\log(1 - D(\hat{x}))$: Khuyến khích Discriminator gán xác suất thấp cho ảnh giả (\hat{x}), tức là $D(\hat{x}) \approx 0$.

Generator (G) cỗ gắng tối thiểu hóa $V(D, G)$ (hoặc tối thiểu hóa L_G): Mục tiêu của Generator là tạo ra ảnh giả sao cho Discriminator không thể phân biệt được với ảnh thật. Context Encoder sử dụng một hàm mất mát tổng hợp:

$$L_G = \lambda \cdot L_{L1}(\mathbf{x}, \hat{\mathbf{x}}) + L_{adv}$$

Trong đó:

- $L_{L1}(\mathbf{x}, \hat{\mathbf{x}})$: Là Mean Absolute Error (MAE) hoặc L1 loss giữa ảnh gốc x và ảnh được tạo ra \hat{x} (thường chỉ tính trên vùng bị che). Nó đo lường sự khác biệt pixel-wise giữa ảnh thật và ảnh giả. Loss này đảm bảo rằng nội dung được điền vào là chính xác về mặt pixel.
- $L_{adv} = -\log(D(\hat{x}))$: Là adversarial loss. Nó khuyến khích Generator tạo ra các vùng ảnh trông **chân thực và sắc nét**, khó phân biệt với ảnh thật bởi Discriminator. Generator cố gắng tối thiểu hóa L_{adv} này, tức là làm cho $D(\hat{x})$ gần 1.
- λ : Là hệ số cân bằng, thường có giá trị $\lambda = 100$ trong Context Encoder gốc, cho thấy sự ưu tiên mạnh mẽ hơn cho việc phục hồi nội dung chính xác (L_{L1}) so với tính chân thực thị giác (L_{adv}).

2.4.7. Quá trình huấn luyện luân phiên

Trong quá trình huấn luyện, Discriminator và Generator được tối ưu hóa luân phiên. Đầu tiên, Discriminator được huấn luyện trên một batch dữ liệu (bao gồm cả ảnh thật và ảnh giả). Sau đó, Generator được huấn luyện để đánh lừa Discriminator. Quá trình này lặp đi lặp lại hàng nghìn, hàng triệu lần cho đến khi mô hình hội tụ.

2.5.Kỹ thuật khôi phục ảnh Image Inpainting

2.5.1. Khái niệm tổng quát

Image Inpainting là kỹ thuật dùng để lấp đầy hoặc khôi phục các vùng bị thiếu, hỏng, hoặc che khuất trong một bức ảnh bằng cách dự đoán nội dung sao cho hài hòa với phần còn lại. Mục tiêu là tạo ra ảnh hoàn chỉnh mà người quan sát không phát hiện ra ảnh từng bị khuyết, đồng thời đảm bảo tính nhất quán về cấu trúc và nội dung.

2.5.2.Các kỹ thuật khôi phục ảnh phổ biến

2.5.2.1.Phương pháp truyền thống (Classical Methods)

- **Texture Synthesis:** tái tạo vùng khuyết bằng cách sao chép các bản vá từ vùng lân cận có kết cấu tương đồng. Ví dụ: Efros & Leung (1999).
- **Diffusion-based (Propagation):** lan truyền thông tin từ biên vùng khuyết vào bên trong bằng phương trình đạo hàm riêng (PDE), ví dụ như Navier–Stokes. Hiệu quả cho đường viền đơn giản nhưng hạn chế với vùng phức tạp.

2.5.2.2.Phương pháp học sâu (Deep Learning-based Methods)

- **Autoencoder / Encoder–Decoder:** nén ảnh bị che thành latent vector rồi giải mã lại toàn bộ ảnh. Dễ bị mờ, thiếu chi tiết.
- **Context Encoder GAN** (Pathak et al., 2016): kết hợp encoder–decoder và GAN. Generator học sinh phần ảnh bị che, Discriminator đánh giá ảnh ghép. Loss bao gồm L1 và adversarial loss.
- **Partial Convolution** (Liu et al., 2018): chỉ áp dụng convolution trên phần ảnh không bị che, đồng thời cập nhật mặt nạ theo từng bước.
- **Gated Convolution + Attention** (Yu et al., 2019): sử dụng gated convolution để điều hướng vùng quan trọng, kết hợp attention để nâng cao khả năng tái lập cấu trúc.
- **Two-stage Inpainting:** giai đoạn coarse sinh ảnh thô, giai đoạn fine refinement

ảnh sắc nét hơn.

2.5.3.Quy trình khôi phục ảnh bằng GAN

Trong phần này chúng ta sẽ áp dụng cho Context Encoder:

1. Tạo mặt nạ che ảnh:

$$x_{\text{masked}} = x \cdot (1 - M) + M \cdot c$$

với M là mặt nạ nhị phân, và c là giá trị mặt nạ (thường là 0).

2. Dự đoán vùng che bằng Generator:

$$\hat{x} = G(x_{\text{masked}})$$

3. Ghép ảnh hoàn chỉnh:

$$x_{\text{reconstructed}} = x \cdot (1 - M) + \hat{x} \cdot M$$

4. Đánh giá bằng Discriminator: phân biệt giữa ảnh thật và ảnh có phần bị phục hồi.

5. Tối ưu hàm mất mát:

$$\mathcal{L}_G = \lambda \cdot \mathcal{L}_{L1}(x, \hat{x}) + \mathcal{L}_{adv}$$

2.5.4.Chiến lược mặt nạ (Masking Strategies)

- **Fixed center mask:** che vùng trung tâm cố định.
- **Random block mask:** che các vùng khối ngẫu nhiên.
- **Free-form mask:** che đường cong, vùng phức tạp, gần với ứng dụng thực tế hơn.

2.5.5.Các yếu tố ảnh hưởng chất lượng phục hồi

- **Kích thước vùng che:** vùng càng lớn, bài toán càng khó.
- **Loại cảnh:** ảnh đơn giản dễ phục hồi hơn ảnh có nhiều vật thể/phức tạp.
- **Hàm mất mát:** lựa chọn loss phù hợp giúp ảnh phục hồi chi tiết và tự nhiên hơn.
- **Kiến trúc mô hình:** Context Encoder, Gated Conv, U-Net... cho kết quả khác nhau.

2.6. Các chỉ số đánh giá chất lượng ảnh

Để đánh giá khách quan chất lượng ảnh được khôi phục bởi mô hình Context Encoder GAN, ta sử dụng tổ hợp các chỉ số phổ biến trong thị giác máy tính, bao gồm cả chỉ số dựa trên pixel, cấu trúc và cảm nhận thị giác:

2.6.1. PSNR (Peak Signal-to-Noise Ratio)

PSNR là một chỉ số phổ biến để đo lường chất lượng khôi phục của ảnh hoặc video bị nén/mất mát. Nó đo mức độ tương đồng giữa ảnh khôi phục và ảnh gốc ở cấp độ pixel và được định nghĩa thông qua MSE (Mean Squared Error). (*sẽ được tìm hiểu ở 2.5.2*)

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right)$$

Trong đó:

- **PSNR:** Tỉ số tín hiệu trên nhiễu (đơn vị: decibel – dB).
- **MAX:** Giá trị pixel lớn nhất trong ảnh (thường là 255 đối với ảnh 8-bit).
- **MSE:** Sai số bình phương trung bình giữa ảnh gốc và ảnh khôi phục.

Ý nghĩa:

- PSNR được đo bằng decibel (dB).
- Giá trị PSNR càng cao càng tốt, cho thấy ảnh được khôi phục/tạo ra có chất lượng tốt hơn và ít nhiễu hơn so với ảnh gốc.
- PSNR từ 30-50 dB thường được coi là chất lượng tốt trong xử lý ảnh kỹ thuật số.

Ưu điểm:

- Được sử dụng rộng rãi và dễ dàng tính toán.
- Có ý nghĩa vật lý rõ ràng về tỷ lệ tín hiệu trên nhiễu.

Nhược điểm:

- Giống như RMSE và MAE, PSNR là một thước đo chất lượng "pixel-wise" và không tương quan mạnh mẽ với hệ thống thị giác của con người. Một ảnh có PSNR cao chưa chắc đã trông đẹp mắt hơn ảnh có PSNR thấp hơn đối với người quan sát.
- Không xem xét các đặc điểm cấu trúc của ảnh.

2.6.2. RMSE (Root Mean Squared Error)

RMSE là một trong những chỉ số đánh giá độ chính xác phổ biến nhất, đo lường sự khác biệt trung bình giữa các giá trị dự đoán (hoặc được tạo ra) và giá trị thực tế (ground truth).

Trong Image Inpainting:

- RMSE đo sai số trung bình bình phương giữa các pixel của ảnh gốc và ảnh khôi phục.
- Giá trị càng thấp càng tốt.

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \hat{x}_i)^2} \quad (1)$$

Trong đó:

- **RMSE**: Sai số gốc trung bình bình phương.
- N : Tổng số pixel trong ảnh.
- x_i : Giá trị pixel tại vị trí i trong ảnh gốc.
- \hat{x}_i : Giá trị pixel tại vị trí i trong ảnh khôi phục.

Ý nghĩa:

- RMSE là một thước đo về độ lớn của sai số. Giá trị RMSE càng nhỏ thì ảnh tạo ra càng giống ảnh gốc.
- Nó có cùng đơn vị với dữ liệu được đo (ví dụ: nếu giá trị pixel từ 0-255, RMSE cũng sẽ ở khoảng đó).
- RMSE nhạy cảm với các sai số lớn hơn vì các sai số được bình phương. Điều này có nghĩa là các sai số lớn sẽ có tác động lớn hơn đến giá trị RMSE.

Ưu điểm:

- Dễ hiểu và dễ tính toán.
- Phổ biến và được sử dụng rộng rãi.

Nhược điểm:

- Không phản ánh tốt chất lượng cảm nhận của con người (perceptual quality). Hai ảnh có cùng RMSE có thể trông rất khác nhau đối với mắt người.
- Chỉ đo lường sai số pixel-wise, không quan tâm đến cấu trúc hoặc ngữ cảnh.
- Nhạy cảm với các outliers (điểm ngoại lai).

2.6.3. MAE (Mean Absolute Error)

MAE là một thước đo khác của sự khác biệt trung bình giữa các giá trị dự đoán và giá trị thực tế. Thay vì bình phương sai số, nó lấy giá trị tuyệt đối của sai số.

Trong Image Inpainting:

- MAE là trung bình của độ lệch tuyệt đối giữa các pixel tương ứng trong ảnh gốc và ảnh sinh.
- Khác với RMSE, MAE không phạt mạnh các lỗi lớn.
- Giá trị càng thấp càng tốt.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |x_i - \hat{x}_i| \quad (2)$$

Trong đó:

- **MAE**: Sai số tuyệt đối trung bình.
- N : Tổng số pixel trong ảnh.
- x_i : Giá trị pixel tại vị trí i trong ảnh gốc.
- \hat{x}_i : Giá trị pixel tại vị trí i trong ảnh khôi phục.

Ý nghĩa:

- Tương tự như RMSE, MAE càng nhỏ thì ảnh tạo ra càng giống ảnh gốc.
- Nó cũng có cùng đơn vị với dữ liệu được đo.
- MAE ít nhạy cảm hơn với các sai số lớn (outliers) so với RMSE vì nó không bình phương sai số.

Ưu điểm:

- Đơn giản, dễ hiểu và dễ tính toán.
- Ít bị ảnh hưởng bởi outliers hơn RMSE.

Nhược điểm:

- Cũng giống như RMSE, MAE chỉ đo lường sai số pixel-wise và không phản ánh tốt chất lượng cảm nhận của con người.
- Không phân biệt được giữa các loại lỗi khác nhau (ví dụ: lỗi mờ, lỗi nhiễu).

2.6.4. SSIM (Structural Similarity Index Measure)

SSIM là một chỉ số được thiết kế để đánh giá chất lượng ảnh dựa trên sự tương đồng về cấu trúc giữa hai ảnh, một yếu tố quan trọng trong cảm nhận của con người. Nó xem xét ba yếu tố chính: độ sáng (luminance), độ tương phản (contrast), và cấu trúc (structure).

Trong Image Inpainting:

- SSIM đánh giá sự tương đồng về **cấu trúc**, **độ sáng**, và **độ tương phản** giữa ảnh gốc và ảnh khôi phục.
- Giá trị nằm trong khoảng $[-1, 1]$; càng gần 1 thì ảnh phục hồi càng “trung thực” về mặt cấu trúc.

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \quad (3)$$

Trong đó:

- **SSIM**: Chỉ số đo mức độ tương đồng về cấu trúc giữa hai ảnh.

- x, y : Hai ảnh đầu vào cần so sánh (ảnh gốc và ảnh khôi phục).

- μ_x, μ_y : Trung bình cường độ pixel của ảnh x và ảnh y .

- σ_x^2, σ_y^2 : Phương sai của ảnh x và y .

- σ_{xy} : Hiệp phương sai giữa x và y .

- C_1, C_2 : Các hằng số ổn định hóa, được tính như:

$$C_1 = (K_1 \cdot L)^2, \quad C_2 = (K_2 \cdot L)^2$$

- $K_1 = 0.01, K_2 = 0.03$: Hằng số mặc định.

- L : Khoảng giá trị động của pixel (ví dụ: 255 với ảnh 8-bit).

Ý nghĩa:

- Giá trị SSIM nằm trong khoảng $[-1, 1]$, với 1 biểu thị sự tương đồng hoàn hảo.

- SSIM được thiết kế để tương quan tốt hơn với cảm nhận chất lượng của con người so với các chỉ số pixel-wise như PSNR.

Ưu điểm:

- Được đánh giá là tương quan tốt hơn với cảm nhận chất lượng chủ quan của con người.
- Xem xét các đặc điểm cấu trúc của ảnh, không chỉ các sai số pixel độc lập.

Nhược điểm:

- Phức tạp hơn trong tính toán so với PSNR.
- Vẫn có thể không hoàn toàn phản ánh chất lượng cảm nhận đối với tất cả các loại ảnh hoặc suy giảm.
- Trong một số trường hợp, nó vẫn có thể bị đánh lừa bởi các thay đổi không đáng kể đối với mắt người.

2.6.5 FID (Fréchet Inception Distance)

FID là một chỉ số đánh giá được sử dụng rộng rãi để đánh giá chất lượng của các mô hình tạo sinh (Generative Models), đặc biệt là GANs. Nó đo lường "khoảng cách" giữa phân phối của các đặc trưng (features) của ảnh thật và phân phối của các đặc trưng của ảnh được tạo ra.

Cách tính:

1. **Trích xuất đặc trưng:** Cả tập dữ liệu ảnh thật và tập dữ liệu ảnh được tạo ra đều được đưa qua một mạng nơ-ron tích chập tiền huấn luyện (thường là mạng Inception-v3, được huấn luyện trên ImageNet). Các đặc trưng (activation) từ một lớp cụ thể (thường là lớp cuối cùng trước lớp phân loại) được trích xuất.

2. **Mô hình hóa phân phối:** Các đặc trưng được trích xuất cho mỗi tập dữ liệu (ảnh thật và ảnh giả) được giả định là tuân theo một phân phối Gaussian đa biến. Hai phân phối này được đặc trưng bởi vector trung bình (μ) và ma trận hiệp phương sai (Σ).

Trong Image Inpainting:

- FID đánh giá độ **thật** và **đa dạng** của ảnh sinh ra bằng cách so sánh phân phối đặc trưng (feature distribution) giữa ảnh gốc và ảnh khôi phục.
- Được tính thông qua đặc trưng trích xuất từ mô hình InceptionV3 huấn luyện trước.
- FID càng thấp thì ảnh sinh càng giống thật và đa dạng.

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2}) \quad (4)$$

Trong đó:

- **FID:** Khoảng cách Fréchet giữa phân phối đặc trưng của ảnh gốc và ảnh sinh.

- μ_r, Σ_r : Trung bình và ma trận hiệp phương sai của đặc trưng trích từ ảnh gốc.
- μ_g, Σ_g : Trung bình và ma trận hiệp phương sai của đặc trưng trích từ ảnh sinh.
- $\|\mu_r - \mu_g\|^2$: Khoảng cách Euclidean bình phương giữa hai vector trung bình.
- Tr : Trace – tổng các phần tử trên đường chéo chính của ma trận.
- $(\Sigma_r \Sigma_g)^{1/2}$: Căn bậc hai ma trận tích.

Ý nghĩa:

- FID càng nhỏ thì càng tốt, cho thấy phân phối của ảnh được tạo ra càng gần với phân phối của ảnh thật.
- Nó không chỉ đo lường sự tương đồng về pixel mà còn về mức độ chân thực và đa dạng của ảnh được tạo ra.
- Một giá trị FID thấp cho thấy ảnh được tạo ra vừa chất lượng cao (realism) vừa đa dạng (diversity) so với ảnh thật.

Ưu điểm:

- Tương quan tốt với chất lượng cảm nhận của con người đối với ảnh tạo ra bởi GANs.
- Đánh giá cả tính chân thực và tính đa dạng của ảnh.
- Ít bị ảnh hưởng bởi "chế độ sụp đổ" (mode collapse) hơn so với các chỉ số như Inception Score (IS).

Nhược điểm:

- Yêu cầu một lượng lớn ảnh để tính toán chính xác (thường là vài nghìn ảnh) để ước tính phân phối Gaussian.
- Phụ thuộc vào mạng trích xuất đặc trưng (Inception-v3). Nếu mạng này không

phù hợp với loại dữ liệu đang xét, kết quả có thể không chính xác.

- Tính toán phức tạp hơn so với các chỉ số pixel-wise.

Tổng kết:

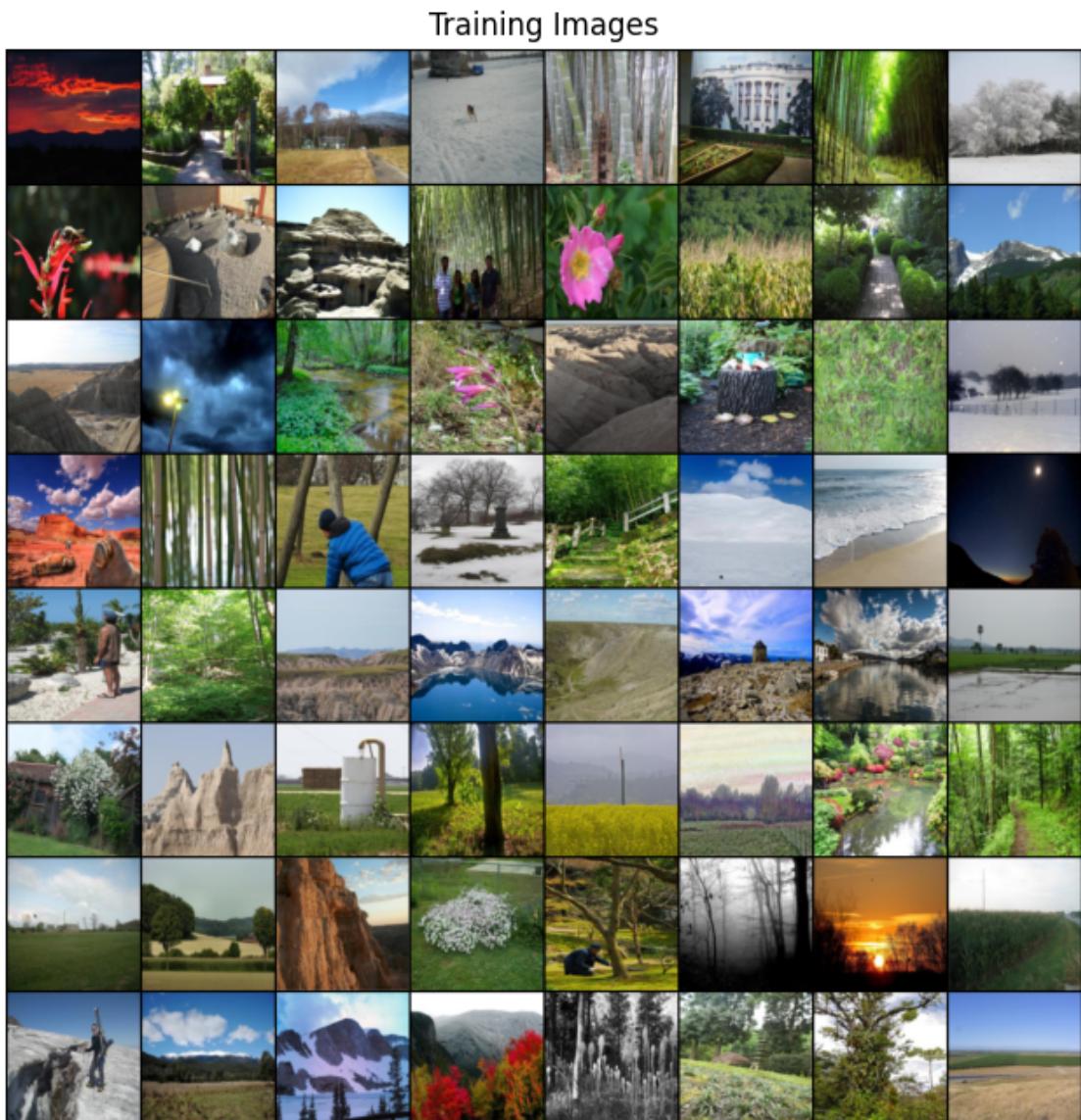
- PSNR, RMSE, MAE: Là các chỉ số "pixel-wise" (point-based). Chúng rất hữu ích để đo lường sai số phục hồi về mặt kỹ thuật, đặc biệt trong các bài toán nơi độ chính xác pixel là quan trọng (ví dụ: nén ảnh không mất mát, khôi phục ảnh có ground truth rõ ràng). Tuy nhiên, chúng không tốt trong việc phản ánh chất lượng cảm nhận của con người. PSNR là thước đo phổ biến nhất trong nhóm này.
- SSIM: Là một chỉ số "structure-based". Nó tốt hơn PSNR/RMSE/MAE trong việc đánh giá chất lượng cảm nhận vì nó xem xét các đặc điểm cấu trúc. Rất thích hợp cho các bài toán khôi phục/nâng cao ảnh nơi cấu trúc là quan trọng.
- FID: Là một chỉ số "feature-based" và "distribution-based". Đây là chỉ số được ưu tiên hàng đầu để đánh giá các mô hình tạo sinh (như GANs) vì nó không chỉ đánh giá tính chân thực của từng ảnh mà còn cả sự đa dạng của toàn bộ tập hợp ảnh được tạo ra, tương quan chặt chẽ với trải nghiệm của con người.

Trong các bài toán như Image Inpainting với Context Encoder, việc kết hợp giữa các chỉ số như PSNR/SSIM (để đánh giá sự phù hợp với ảnh gốc) và FID (để đánh giá tính chân thực và đa dạng của vùng được tạo ra) là cách tốt nhất để có cái nhìn toàn diện về hiệu suất của mô hình.

Chương III: Thực nghiệm

3.1. Dữ liệu đầu vào

- Tập dữ liệu sử dụng trong đồ án bao gồm 20 nhãn được tách ra từ tập place365-small với mỗi nhãn bao gồm 5000 ảnh với mỗi ảnh có kích thước 256x256.
- Trong bài toán này, ta lấy các ảnh ngẫu nhiên từ mỗi nhãn nhằm đa dạng hóa ngũ cảnh bài toán



Hình 0.1: Các ảnh sử dụng trong việc training mô hình

3.2. Cài đặt mô hình

3.2.1. Cài đặt thư viện

- Đầu tiên ta khai báo các thư viện sẽ sử dụng cho mô hình

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import torch.nn.functional as F
import gc
```

Hình 0.2: Những thư viện sử dụng trong bài toán

Trong mô hình mạng GAN, nhóm chúng em sử dụng thư viện PyTorch hỗ trợ cho việc xây dựng mô hình cũng như xử lý tính toán. Đồng thời nhóm chúng em cũng sử dụng một số các thư viện khác nhằm hỗ trợ cho việc trực quan hóa mô hình, bổ trợ tính toán cũng như tối ưu hóa bộ nhớ nhằm giúp mô hình thực hiện các tác vụ tính toán dễ dàng hơn.

3.2.2. Thiết lập tham số mô hình và xử lý dữ liệu

a, 3.2.2.1. Thiết lập tham số mô hình

Sau khi cài đặt xong thư viện sử dụng cho mô hình, ta sẽ thiết lập các tham số cho mô hình

```
# Root directory for dataset
dataroot = 'Data/dataset'

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 256

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 128

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Weight between various components of total generator loss
weightRecon = 0.99

# Number of training epochs
num_epochs = 100

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

Hình 0.3: Cấu hình tham số mô hình

- Đầu tiên ta khai báo đường dẫn tới dataset của mô hình.
- Sau đó ta sẽ cấu hình số luồng tải dữ liệu và kích thước batch size ta sẽ sử dụng:
 - workers = 2: Số lượng tiến trình con (workers) dùng để tải dữ liệu, giúp tăng tốc độ tải dữ liệu.
 - batch_size = 256: Số lượng mẫu dữ liệu trong mỗi batch được tải lên để huấn luyện.

- Sau đó ta sẽ khai báo các tham số sử dụng trong mô hình:
 - image_size = 128: Đây là kích thước không gian (chiều dài và chiều rộng) của ảnh đầu vào và đầu ra. Tất cả các ảnh sẽ được resize về kích thước 128x128 trước khi đưa vào mô hình.
 - nc = 3: Number of channels: Số kênh màu của ảnh. Ta sử dụng ảnh màu RGB có 3 kênh: Red, Green, Blue.
 - nz = 100: Kích thước vector nhiễu (latent vector) – đầu vào cho Generator. Generator sẽ nhận một vector ngẫu nhiên kích thước 100, và cô gắng "biến hóa" nó thành một ảnh đầy đủ.
 - ngf = 64: (Number of Generator Feature maps) Là số lượng feature maps (bộ lọc convolution) trong lớp đầu tiên của Generator. Mỗi lớp tiếp theo thường sẽ tăng dần số lượng kênh (ví dụ: 64 → 128 → 256 → 512). Tham số này ảnh hưởng trực tiếp đến sức mạnh biểu diễn và độ phức tạp của Generator.
 - ndf = 64: (Number of Discriminator Feature maps) Tương tự như ngf, nhưng dùng cho Discriminator. Là số lượng bộ lọc của lớp đầu tiên trong Discriminator. Discriminator sẽ giảm dần kích thước không gian và tăng số kênh để xác định xem ảnh là thật hay giả.
- Cuối cùng ta khai báo các siêu tham số điều khiển quá trình huấn luyện mô hình:
 - weightRecon = 0.99: Đây là trọng số của loss tái tạo (reconstruction loss) trong total generator loss. Với weightRecon = 0.99, mô hình sẽ ưu tiên việc tái tạo lại hình ảnh sát với ảnh thật hơn là việc chỉ đánh lừa Discriminator.
 - num_epochs = 100: Là số vòng lặp toàn bộ tập dữ liệu huấn luyện.
 - lr = 0.0002: Tốc độ học (learning rate) dùng trong tối ưu hóa (Adam Optimizer). Đây là một giá trị nhỏ để đảm bảo mô hình học từ từ, tránh bị dao động hoặc sai lệch nghiêm trọng trong cập nhật trọng số.
 - beta1 = 0.5: Đây là siêu tham số cho Adam Optimizer, cụ thể là hệ số momen đầu tiên. beta1 kiểm soát mức độ ảnh hưởng của gradient trung bình trong quá khứ lên cập nhật hiện tại.
 - ngpu = 1: Số lượng GPU có sẵn để sử dụng cho huấn luyện.
- Sau đó ta sẽ cài đặt số seed ngẫu nhiên nhằm đảm bảo tính tái lập (reproducibility) trong quá trình huấn luyện mô hình với PyTorch.

```

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)

```

Hình 0.4: Cài đặt seed

3.2.2.2. Xử lý dữ liệu

Đầu tiên ta sẽ tạo dataset và dataloader để tải dữ liệu lên

```

# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))

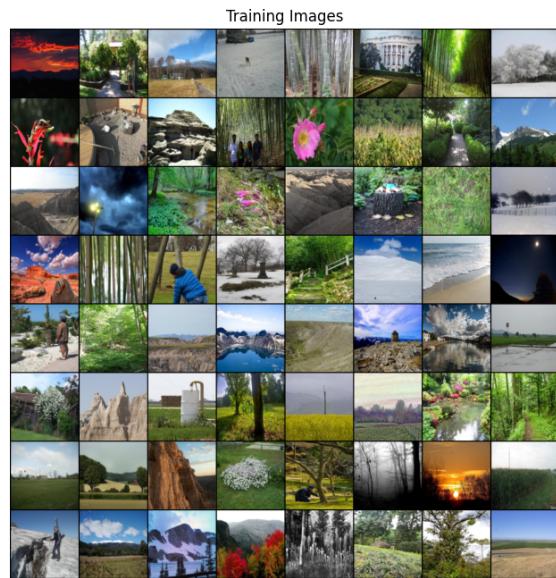
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers,
                                         pin_memory=False, # Disable pinned memory
                                         persistent_workers=False # Don't keep workers alive between iterations
                                         )

# Decide which device we want to run on
device = torch.device("cuda" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
print(device)
# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(), (1,2,0)))

```

Hình 0.5: Tải dữ liệu

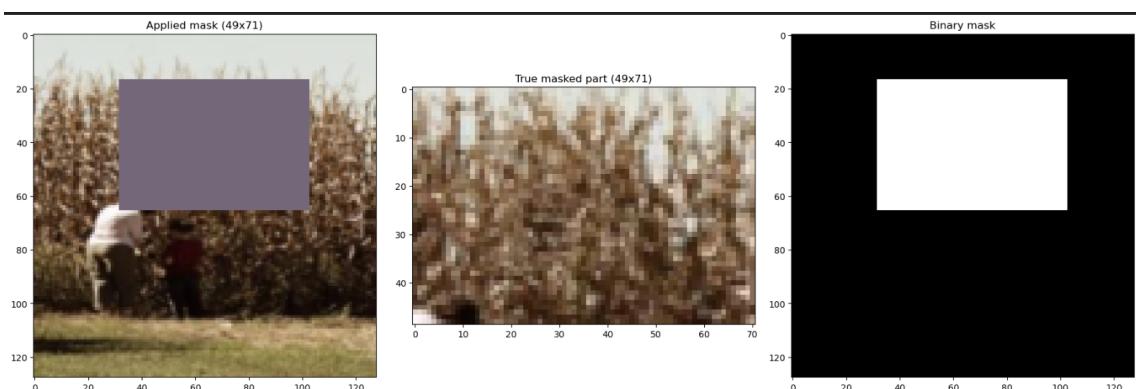
- Đầu tiên ta load dataset từ thư mục ảnh. Sử dụng ImageFolder để load ảnh từ một thư mục. dataroot là đường dẫn đến thư mục chứa các ảnh huấn luyện. Ảnh được nạp sẽ được chuyển qua chuỗi biến đổi (transforms) để chuẩn hóa trước khi huấn luyện. Sau đó ta sẽ Resize(image_size) → Phóng hoặc thu ảnh về kích thước vuông $\text{image_size} \times \text{image_size}$. CenterCrop(image_size) → Cắt ảnh chính giữa về kích thước $\text{image_size} \times \text{image_size}$. ToTensor() → Chuyển ảnh từ dạng PIL/numpy thành Tensor với shape (C, H, W) và chuẩn hóa giá trị pixel về [0, 1]. Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) → Chuẩn hóa giá trị pixel từ [0,1] sang [-1,1], để phù hợp với GAN loss hoạt động tốt hơn.



Hình 0.6: Một batch ảnh huấn luyện

- Tiếp đó ta tạo DataLoader để tải dữ liệu theo batch với các tham số sau:
 - batch_size: số lượng ảnh trong mỗi batch.
 - shuffle=True: trộn dữ liệu trước khi huấn luyện
 - num_workers: số tiến trình đọc dữ liệu song song.
 - pin_memory=False: tắt việc pin bộ nhớ để tăng tương thích (hữu ích nếu CPU/GPU không đồng bộ tốt).
 - persistent_workers=False: không giữ tiến trình worker tồn tại giữa các epoch (giảm RAM nếu máy yếu).
- Cuối cùng ta chọn thiết bị huấn luyện và hiển thị một batch ảnh huấn luyện.

Để phục vụ cho việc tái tạo lại ảnh đã bị khuyết, đầu tiên ta cần tạo ra phần khuyết trong bức ảnh từ đó mô hình có thể tái tạo ra phần ảnh mới.



Hình 0.7: Ảnh sau khi cắt 1 phần

3.2.3. Cài đặt kiến trúc mô hình

3.2.3.1. Cài đặt Generator

Generator này có kiến trúc Encoder-Decoder với các kết nối tắt (skip connections), tương tự như kiến trúc U-Net. Kiến trúc này rất hiệu quả trong các tác vụ chuyển đổi ảnh sang ảnh (image-to-image translation) vì nó cho phép thông tin từ các tầng nông hơn (chi tiết cấp thấp) của bộ mã hóa được truyền trực tiếp sang các tầng sâu hơn của bộ giải mã, giúp tái tạo chi tiết tốt hơn.

- Bộ mã hóa (Encoder):

- Đầu vào: Ảnh màu 3 kênh (ví dụ: RGB).
 - Cấu trúc: Gồm 5 khối tuần tự (enc1 đến enc5):
 - * Mỗi khối sử dụng lớp nn.Conv2d (tích chập 2D) với kernel_size=4, stride=2, padding=1. Cấu hình này làm giảm kích thước không gian của bản đồ đặc trưng đi một nửa ở mỗi khối.
 - * Số lượng kênh đặc trưng tăng dần: $3 \rightarrow 64 \rightarrow 64 \rightarrow 128 \rightarrow 256 \rightarrow 512$.
 - * Hàm kích hoạt nn.LeakyReLU(0.2, inplace=True) được sử dụng để tránh hiện tượng "ReLU chết" và cho phép một lượng nhỏ gradient đi qua ngay cả khi đầu vào âm.
 - * Lớp nn.BatchNorm2d (chuẩn hóa theo batch) được sử dụng từ enc2 trở đi để ổn định quá trình huấn luyện và giúp mô hình hội tụ nhanh hơn.
 - Lớp cổ chai (Bottleneck): Đây là lớp trung gian giữa bộ mã hóa và bộ giải mã, nơi biểu diễn đặc trưng đạt mức độ nén cao nhất.
 - Cấu trúc: Một khối tuần tự (bottleneck):
 - * Sử dụng nn.Conv2d với kernel_size=4, stride=1, padding=0. Với kernel_size=4 và không có padding, lớp này sẽ giảm đáng kể kích thước không gian (ví dụ: từ 4x4 xuống 1x1).
 - * Số kênh đầu ra là 2048.
 - * Sử dụng nn.BatchNorm2d và nn.LeakyReLU.
 - Bộ giải mã (Decoder) với Kết nối tắt: Bộ giải mã có nhiệm vụ khôi phục lại ảnh từ biểu diễn đặc trưng tiềm ẩn, đồng thời sử dụng thông tin từ các kết nối tắt để giữ lại các chi tiết.
 - Cấu trúc: Gồm 5 khối tuần tự (dec1 đến dec5):

```

ImprovedGenerator(
    (enc1): Sequential(
        (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (enc2): Sequential(
        (0): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (enc3): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (enc4): Sequential(
        (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (enc5): Sequential(
        (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (bottleneck): Sequential(
        (0): Conv2d(512, 2048, kernel_size=(4, 4), stride=(1, 1), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (dec1): Sequential(
        (0): ConvTranspose2d(2048, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Dropout(p=0.2, inplace=False)
    )
    (dec2): Sequential(
        (0): ConvTranspose2d(1024, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
    )
    (dec3): Sequential[]
        (0): ConvTranspose2d(512, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
    ]
    (dec4): Sequential(
        (0): ConvTranspose2d(256, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
    )
    (dec5): Sequential(
        (0): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): Tanh()
    )
)
)

```

Hình 0.8: Kiến trúc Generator

- * Mỗi khối sử dụng lớp nn.ConvTranspose2d (tích chập chuyển vị hay "deconvolution") để tăng kích thước không gian của bản đồ đặc trưng.
 - dec1: kernel_size=4, stride=1, padding=0.
 - dec2 đến dec5: kernel_size=4, stride=2, padding=1. Cấu hình này làm tăng kích thước không gian lên gấp đôi ở mỗi khối.
- * Kết nối tắt (Skip Connections): Đây là một cải tiến quan trọng. Đầu ra của mỗi tầng trong bộ mã hóa (e1, e2, e3, e4, e5) được nối (concatenate) với đầu vào của tầng tương ứng trong bộ giải mã. Điều này giúp bộ giải mã truy cập được các đặc trưng cấp thấp (ví dụ: cạnh, kết cấu) từ bộ mã hóa, rất quan trọng cho việc tái tạo ảnh chất lượng cao.
 - dec2 nhận đầu vào từ d1 (đầu ra dec1) nối với e5.
 - dec3 nhận đầu vào từ d2 nối với e4.
 - dec4 nhận đầu vào từ d3 nối với e3.
 - ec5 nhận đầu vào từ d4 nối với e2.
- * Số lượng kênh đặc trưng giảm dần: 2048 → 512 (sau dec1) → 256 → 128 → 64 → 3.
- * Hàm kích hoạt nn.ReLU(inplace=True) được sử dụng trong các khôi giải mã, ngoại trừ lớp cuối cùng.
- * Lớp nn.BatchNorm2d được sử dụng từ dec1 đến dec4.
- * nn.Dropout(0.2) được thêm vào dec1 để chính quy hóa, giúp mô hình khái quát hóa tốt hơn và tránh overfitting.
- * Lớp đầu ra (dec5): Tạo ra ảnh 3 kênh và sử dụng hàm kích hoạt nn.Tanh(). Hàm Tanh sẽ đưa các giá trị pixel đầu ra về khoảng [-1, 1], phù hợp với nhiều phương pháp chuẩn hóa ảnh.

3.2.3.2. Cài đặt Discriminator

Bộ phân biệt này là một mạng tích chập sâu (Deep Convolutional Neural Network - DCNN) được huấn luyện để phân loại xem một bức ảnh (hoặc một vùng trong ảnh) là thật hay giả (được tạo ra bởi bộ sinh). Kiến trúc này lấy cảm hứng từ PatchGAN, nơi đầu ra không phải là một giá trị xác suất duy nhất cho toàn bộ ảnh, mà là một bản đồ đặc trưng 2D, trong đó mỗi phần tử đánh giá "tính chân thực" của một "mảng" (patch) trong ảnh đầu vào.

```

ImprovedDiscriminator(
    (layer1): Sequential(
        (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (layer2): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
        (3): Dropout2d(p=0.25, inplace=False)
    )
    (attention): SelfAttention(
        (query_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
        (key_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
        (value_conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (softmax): Softmax(dim=-1)
    )
    (layer3): Sequential[
        (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    ]
    (layer4): Sequential(
        (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (layer5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (sigmoid): Sigmoid()
)

```

Hình 0.9: Kiến trúc Discriminator

- Đầu vào: Ảnh màu 3 kênh (ảnh thật hoặc ảnh do bộ sinh tạo ra).
- Các lớp tích chập:
 - layer1: Conv2d (3 kênh → 64 kênh), kernel_size=4, stride=2, padding=1.
Không có BatchNorm2d ở lớp đầu tiên (một thực hành phổ biến). Sử dụng LeakyReLU.
 - layer2: Conv2d (64 kênh → 128 kênh), kernel_size=4, stride=2, padding=1.
Có BatchNorm2d, LeakyReLU và Dropout2d(0.25) để chính quy hóa.
 - attention: Lớp Tự chú ý (Self-Attention). Đây là một cải tiến đáng kể. Lớp SelfAttention(128) cho phép mô hình tập trung vào các vùng quan trọng hơn trong bản đồ đặc trưng. Nó giúp mô hình hiểu được các mối quan hệ xa trong ảnh, thay vì chỉ tập trung vào các vùng lân cận cục bộ.
 - layer3: Conv2d (128 kênh → 256 kênh), kernel_size=4, stride=2, padding=1.
Có BatchNorm2d và LeakyReLU.
 - layer4: Conv2d (256 kênh → 512 kênh), kernel_size=4, stride=2, padding=1.
Có BatchNorm2d và LeakyReLU.

- Lớp phân loại cuối cùng (layer5):
 - Conv2d (512 kênh → 1 kênh), kernel_size=4, bias=False. Lớp này giảm số kênh xuống còn 1, đại diện cho điểm số "thật/giả". Kernel size là 4 (và stride mặc định là 1, padding là 0) sẽ tiếp tục giảm kích thước không gian. Nếu đầu vào của lớp này là $H \times W$, thì đầu ra sẽ là $(H4+1) \times (W4+1)$.
- Hàm kích hoạt Sigmoid (sigmoid):
 - nn.Sigmoid() được áp dụng cho đầu ra của layer5. Hàm sigmoid sẽ đưa các giá trị điểm số về khoảng [0, 1], có thể được diễn giải là xác suất ảnh là thật.

3.2.4. Thiết lập cấu hình huấn luyện mạng GAN

- Ta khởi tạo các danh sách để theo dõi tiến trình

```
# Training Loop
# Lists to keep track of progress
img_list = []
G_losses = []
G_mse_losses = []
G_Tot_losses = []
D_losses = []
iters = 0
```

Hình 0.10: Các danh sách để theo dõi tiến trình

- Sau đó ta lặp qua mỗi epoch (một lượt duyệt toàn bộ tập dữ liệu) và trong mỗi epoch ta lặp qua mỗi batch trong DataLoader:

```

#####
# (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
#####
## Train with all-real batch
netD.zero_grad()
# Format batch
real_cpu = data[0].to(device)
b_size = real_cpu.size(0)
label = torch.full((b_size,), real_label, dtype=torch.float, device=device)

masked_samples, true_masked_tensor, mask_info = get_random_block_mask(real_cpu)

# Forward pass real batch through D
output = netD(true_masked_tensor).view(-1)
# Calculate loss on all-real batch
errD_real = criterion(output, label)
# Calculate gradients for D in backward pass
errD_real.backward()
D_x = output.mean().item()

## Train with all-fake batch
# Generate fake image batch with G
fake = netG(masked_samples)
label.fill_(fake_label)
# Classify all fake batch with D
output = netD(fake.detach()).view(-1)
# Calculate D's loss on the all-fake batch
errD_fake = criterion(output, label)
# Calculate the gradients for this batch, accumulated (summed) with previous gradients
errD_fake.backward()
D_G_z1 = output.mean().item()

# Compute error of D as sum over the fake and the real batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()

```

Hình 0.11: Cập nhật mạng Discriminator

- Hàm tối đa hóa $\log(D(x)) + \log(1 - D(G(z)))$. Mục tiêu của bước này là huấn luyện Discriminator (netD) để nó phân biệt tốt hơn giữa ảnh thật (x) và ảnh giả (G(z)) do Generator tạo ra.
- Huấn luyện với Ảnh Thật:
 - * netD.zero_grad(): Xóa gradient cũ của Discriminator.
 - * Lấy batch ảnh thật (real_cpu) từ dataloader.
 - * Tạo nhãn "thật" (real_label, thường là 1.0).
 - * Chuẩn bị dữ liệu cho Context Encoder: Tạo masked_samples (ảnh bị che) và true_masked_tensor (ảnh gốc không bị che) từ real_cpu.
 - * Đưa true_masked_tensor qua netD để nhận output.
 - * Tính errD_real (loss của D trên ảnh thật) bằng criterion (ví dụ: BCELoss).

- * errD_real.backward(): Tính gradient cho D từ ảnh thật.
- * Lưu D_x (output trung bình của D cho ảnh thật).
- Huấn luyện vớiẢnh Giả:
 - * Tạo batch ảnh giả (fake) bằng cách cho masked_samples qua netG.
 - * Tạo nhãn "giả" (fake_label, thường là 0.0).
 - * Đưa fake.detach() (ảnh giả, tách khỏi đồ thị tính toán của G) qua netD.
 - * Tính errD_fake (loss của D trên ảnh giả).
 - * errD_fake.backward(): Tính gradient cho D từ ảnh giả (cộng dồn với gradient từ ảnh thật).
 - * Lưu D_G_z1 (output trung bình của D cho ảnh giả, trước khi D cập nhật).
- Cập nhật Discriminator:
 - * errD = errD_real + errD_fake: Tính tổng loss cho D.
 - * optimizerD.step(): Cập nhật trọng số của netD.

```

#####
# (2) Update G network: maximize log(D(G(z)))
#####
netG.zero_grad()
label.fill_(real_label) # fake labels are real for generator cost

# Since we just updated D, perform another forward pass of all-fake batch through D
output = netD(fake).view(-1)
# Calculate G's loss based on this output
errG = criterion(output, label)
D_G_z2 = output.mean().item()

# Add MSE loss between generated infill and original
l2_weights = get_weights(fake.size())
lossGeneratorRecon = weighted_mse_loss(fake, true_masked_tensor, l2_weights.to(device))
lossGeneratorTotal = (1 - weightRecon) * errG + weightRecon * lossGeneratorRecon

lossGeneratorTotal.backward()
optimizerG.step()

```

Hình 0.12: Cập nhật mạng Generator

- Mục tiêu của bước này là huấn luyện Generator (netG) để nó tạo ra những ảnh giả có thể đánh lừa Discriminator, khiến Discriminator phân loại chúng là "thật".
- Chuẩn bị:
 - netG.zero_grad(): Xóa gradient cũ của Generator.

- Đặt label thành real_label (vì G muốn D nghĩ ảnh giả là thật).
- Tính Loss Đồi Nghịch (Adversarial Loss):
 - Đưa batch ảnh giả (fake - lần này không .detach()) qua netD (phiên bản D vừa được cập nhật).
 - Tính errG (loss đối nghịch của G) bằng criterion.
 - Lưu D_G_z2 (output trung bình của D cho ảnh giả, sau khi D cập nhật).
- Tính Loss Tái Tạo (Reconstruction Loss - cho Context Encoder): Tính lossGeneratorRecon (ví dụ: weighted_ms_loss) giữa fake (ảnh G tạo ra) và true_masked_tensor (ảnh gốc không bị che).
- Tính Tổng Loss và Cập nhật Generator:
 - lossGeneratorTotal = (1 - weightRecon) * errG + weightRecon * lossGeneratorRecon: Kết hợp loss đối nghịch và loss tái tạo.
 - lossGeneratorTotal.backward(): Tính gradient cho G.
 - optimizerG.step(): Cập nhật trọng số của netG.

```

# Output training stats
if i % 50 == 0:
    print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
          % (epoch, num_epochs, i, len(dataloader),
              errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

    # Save Losses for plotting later
    G_losses.append(errG.item())
    G_mse_losses.append(lossGeneratorRecon.item())
    G_Tot_losses.append(lossGeneratorTotal.item())
    D_losses.append(errD.item())

# Check how the generator is doing by saving G's output on fixed examples
if (iters % 50 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
    with torch.no_grad():
        fake = netG(viz_masked_samples).detach()
        viz_infill = viz_masked_samples.clone()

        # Correctly apply the generated content to the masked areas
        # We need to resize the 64x64 outputs to match the original mask sizes
        for idx, (top, left, mask_h, mask_w) in enumerate(viz_mask_info):
            if idx < fake.size(0): # Make sure we don't exceed batch size
                # Resize the generated 64x64 patch to the actual mask size
                resized_fake = F.interpolate([
                    fake[idx].unsqueeze(0),
                    size=(mask_h, mask_w),
                    mode='bilinear',
                    align_corners=False
                ])

                # Place the resized content into the correct position
                viz_infill[idx, :, top:top+mask_h, left:left+mask_w] = resized_fake.squeeze(0)

    img_list.append(vutils.make_grid(viz_infill[:64].cpu(), padding=2, normalize=True))

    iters += 1

```

Hình 0.13: In ra các chỉ số huấn luyện và Kiểm tra generator

- Mục tiêu của bước này nhằm in ra các chỉ số huấn luyện và kiểm tra xem generator đang hoạt động như thế nào bằng cách lưu output của G trên các ví dụ cố định.
- Ghi nhận thông tin:
 - Nếu $i \% 50 == 0$ (ví dụ, mỗi 50 batch)
 - * In ra các giá trị loss (errD, errG) và output của D (D_x , D_G_z1 , D_G_z2).
 - * Lưu các giá trị loss vào danh sách tương ứng (G_losses , D_losses , v.v.).
- Lưu ảnh mẫu:
 - Nếu $iters \% 50 == 0$ hoặc ở cuối quá trình huấn luyện:
 - * Sử dụng `torch.no_grad()` để không tính gradient.

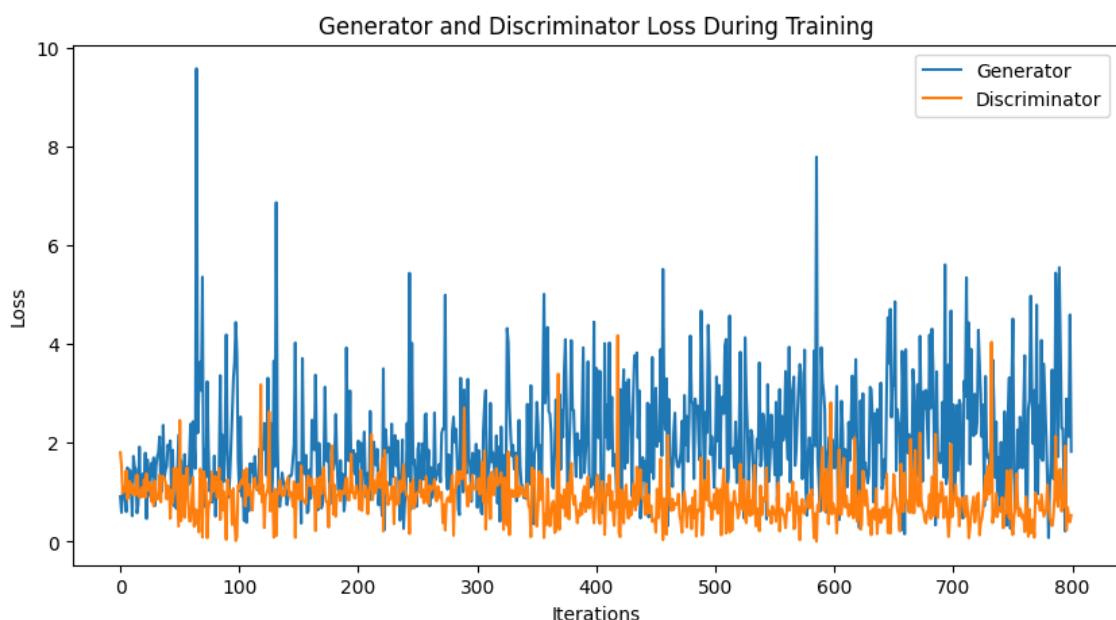
- * Cho viz_masked_samples (một tập mẫu bị che cố định) qua netG để tạo fake_viz.
 - * Ghép ảnh: Thay đổi kích thước fake_viz (nếu cần, ví dụ dùng F.interpolate) và đặt vào đúng vị trí trên viz_infill (bản sao của viz_masked_samples) dựa trên viz_mask_info.
 - * Lưu lối ảnh kết quả vào img_list.
- Cập nhật bộ đếm: iters += 1: Tăng bộ đếm tổng số batch

3.3. Kết quả thực nghiệm

Sau khi training mô hình xong, ta sẽ lấy 1 batch sau khi đã qua xử lý để thấy kết quả sau khi huấn luyện mô hình.



Hình 0.14: Kết quả sau khi huấn luyện



Hình 0.15: Biểu đồ giá trị mất mát của mô hình khi huấn luyện

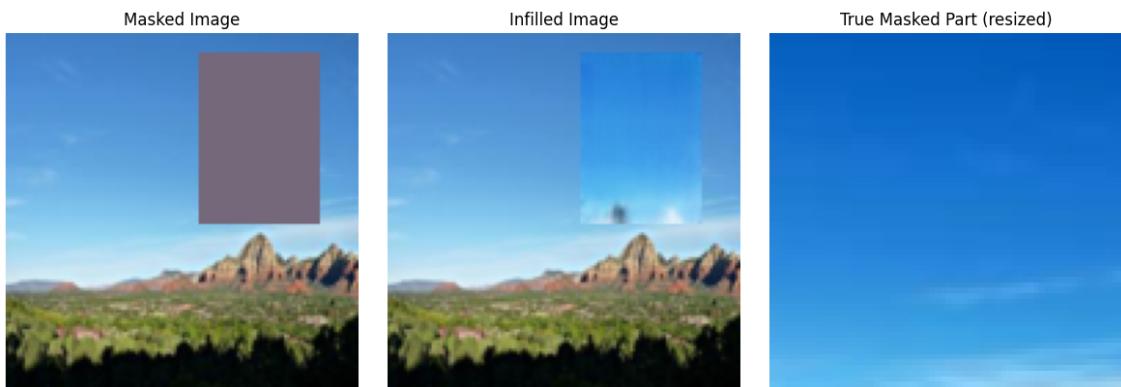
- Qua quá trình huấn luyện mô hình GAN, ta có thể thấy rằng:
 - Về Generator Loss:
 - * Loss có xu hướng dao động mạnh và không ổn định trong suốt quá trình training.
 - * Giá trị loss tương đối cao (dao động từ 2-9) và không có xu hướng giảm rõ ràng theo thời gian.
 - * Xuất hiện nhiều spike đột ngột, đặc biệt ở iteration 60 và 580.

- Về Discriminator Loss:
 - * Loss tương đối ổn định hơn Generator, dao động quanh mức 0.5-2.0.
 - * Có xu hướng convergence tốt hơn so với Generator.
 - * Ít có những biến động đột ngột và extreme spikes.
- Về sự cân bằng giữa hai networks:
 - * Generator và Discriminator chưa đạt được trạng thái cân bằng (Nash equilibrium).
 - * Generator dường như đang "thua" trong cuộc đua adversarial, thể hiện qua loss cao và không ổn định.
 - * Discriminator có vẻ "mạnh" hơn, có thể đang phân biệt real/fake quá tốt.

{'PSNR': 26.139890670776367, 'MAE': 0.015355143696069717, 'RMSE': 0.05358882248401642, 'SSIM': 0.9559739828109741, 'LPIPS-like': 0.013134032487869263, 'FID': 50.78541846874184}

Hình 0.16: Các chỉ số đánh giá mô hình

- Dựa trên các chỉ số đánh giá, ta có những nhận xét về hiệu suất của mô hình như sau:
 - SSIM = 0.956 - Rất tốt, cho thấy mô hình bảo toàn được cấu trúc và texture của ảnh gốc.
 - MAE = 0.015 - Thấp, chứng tỏ sai số trung bình giữa pixel dự đoán và thực tế nhỏ.
 - RMSE = 0.054 - Tương đối thấp, cho thấy độ chính xác pixel-wise khá tốt.
 - LPIPS-like = 0.013 - Rất thấp, có nghĩa là sự khác biệt perceptual giữa ảnh sinh ra và ảnh gốc nhỏ.
 - PSNR = 26.14 dB - Ở mức trung bình, có thể tốt hơn (thường mong muốn >30dB)
 - FID = 50.79 - Tương đối cao, cho thấy chất lượng ảnh sinh ra chưa thực sự tự nhiên
- Dựa trên kết quả trên, ta có những nhận xét về hiệu suất của mô hình:
 - Điểm mạnh của mô hình:
 - * Tính liên tục tốt - Vùng được fill khớp tự nhiên với background xung quanh.



Hình 0.17: Thử nghiệm trên một ảnh ngẫu nhiên

- * Màu sắc hài hòa.
- * Không có artifacts rõ ràng - Không thấy boundary effects hay discontinuities.
- * Semantic understanding - Mô hình hiểu được đây là khu vực xung quanh và tạo ra nội dung phù hợp.
- Điểm yếu cần cải thiện:
 - * Thiếu chi tiết - Vùng được fill khá smooth và đơn điệu, không có texture phong phú.
 - * Ảnh được tạo ra còn thiếu tự nhiên.
 - * Hơi blur - Vùng inpainted có vẻ mờ hơn so với phần gốc của ảnh.
 - * Không tái tạo được các chi tiết nhỏ như sự đa dạng trong bầu trời.
- Kết luận: Mô hình hoạt động tốt cho task cơ bản - tạo ra kết quả có thể chấp nhận được.

LỜI BẠT

Qua quá trình thực hiện báo cáo này, chúng em đã có cơ hội tìm hiểu sâu hơn về tái tạo ảnh sử dụng mạng GAN, từ đó ứng dụng được cách khai triển vào các bài toán khác nhau. Nếu có gì sai sót, chúng em xin được nhận lời góp ý, phê bình của cô và cố gắng để hoàn thiện hơn trong các bài báo cáo tương lai.

TÀI LIỆU THAM KHẢO

<http://places2.csail.mit.edu/PAMIplaces.pdf>

<http://places.csail.mit.edu/placesNIPS14.pdf>

<https://viblo.asia/p/deep-learning-tim-hieu-ve-mang-tich-chap-cnn-maGK73bOKj2>

<https://viblo.asia/p/bai-1-mang-gan-mang-dem-lai-su-sang-tao-cho-tri-tue-nhan-tao-maGK7VbD5j2>

<https://nttuan8.com/bai-1-gioi-thieu-ve-gan/>