

AGA U5

Conditional Generative Adversarial Networks (CGANs) - Adding Control

The fundamental limitation of a vanilla GAN is that it's an unsupervised model. It learns the underlying data distribution and generates samples from it, but you have no control over what kind of sample it generates. For example, if you train a GAN on a dataset of MNIST digits, it will generate random digits (0-9), but you can't tell it to generate specifically a '3' or an '8'.

CGANs address this by introducing a "condition" or "label" (y) to both the Generator and the Discriminator. This condition provides contextual information, allowing the model to generate specific types of outputs.

The Key Idea: Both the Generator and the Discriminator receive the condition y as an additional input, along with their usual inputs (noise for G, data for D).

1. The Conditional Generator (G): The Directed Counterfeiter

- **Input:** Instead of just a random noise vector z , the Conditional Generator takes both the random noise z and the condition y as input.
- **How y is incorporated:** The condition y (e.g., a class label, an image, a text description) is typically embedded into a vector space and then concatenated with the noise vector z . This combined vector is then fed into the Generator network.
- **Output:** Generates a synthetic data sample $G(z, y)$ that is conditioned on y . For example, if y is the label '3', the generator aims to produce an image of a '3'.
- **Goal:** To generate data that is both realistic and consistent with the provided condition y , to fool the Discriminator.

2. The Conditional Discriminator (D): The Context-Aware Detective

- **Input:**
 - Receives a real data sample x and its corresponding condition y_x .
 - Receives a fake data sample $G(z, y_g)$ (generated by the Conditional Generator) and the condition y_g that was used to generate it.
- **How y is incorporated:** Similar to the Generator, the condition y is often concatenated with the data input (either real x or fake $G(z, y)$) and fed into the Discriminator network.
- **Output:** A single scalar probability $D(x, y)$ or $D(G(z, y), y)$, indicating how "real" the input data is, given the provided condition.
- **Goal:** To distinguish between real and fake data, and to verify that the generated data matches the provided condition. If the Generator produces a dog image but is told to generate a cat, the Discriminator should ideally classify it as fake.

3. The Conditional Adversarial Training Process:

The training process is very similar to a vanilla GAN, but with the added condition y always present.

- **Step 1: Conditional Discriminator Training**
 - **Real Data:** The Discriminator receives a real data sample x from the training set along with its true label/condition y_x . It is trained to output a high probability (close to 1) for this pair (x, y_x) .
 - **Fake Data:** The Generator takes a random noise z and a randomly chosen condition y_g (e.g., a random digit label) to generate $G(z, y_g)$. The Discriminator then receives this generated sample along with the condition y_g used to create it. It is trained to output a low probability (close to 0) for this pair $(G(z, y_g), y_g)$.
 - The Discriminator's weights are updated to maximize its ability to correctly classify (x, y_x) as real and $(G(z, y_g), y_g)$ as fake.

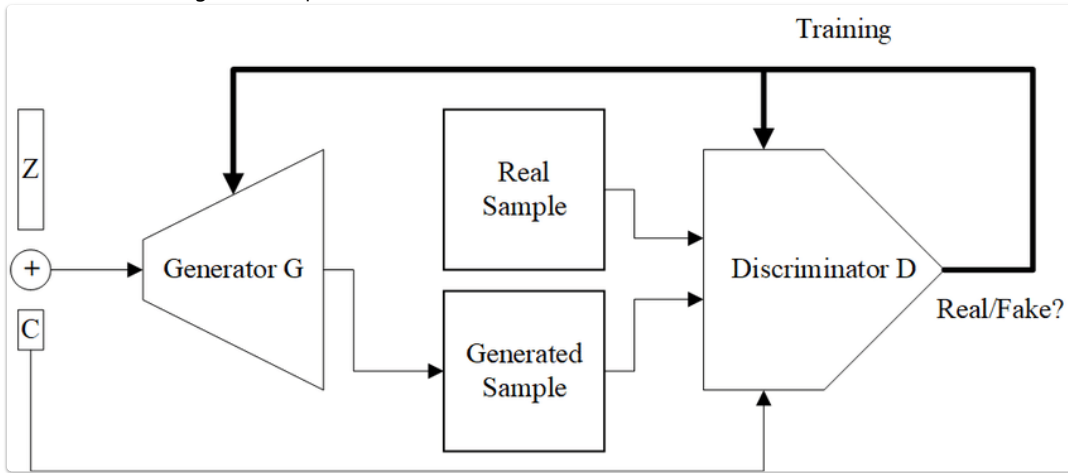
- **Step 2: Conditional Generator Training**

The Generator takes a random noise z and a randomly chosen condition y_g to generate $G(z, y_g)$.

This generated sample $G(z, y_g)$ and its corresponding condition y_g are fed to the Discriminator.

* The Generator's weights are updated to maximize the probability that the Discriminator classifies $(G(z, y_g), y_g)$ as "real." The

Discriminator's weights are kept fixed.



Mathematical Formulation (Loss Functions):

The objective function for a standard GAN is often formulated as a minimax game:

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

For a **Conditional GAN**, the condition y is simply incorporated into both the Discriminator and Generator inputs:

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

Here:

- x is a real data sample.
- y is the conditional information (e.g., class label, text description, another image).
- $p_{data}(x)$ is the real data distribution.
- $p_z(z)$ is the noise distribution.
- $G(z|y)$ denotes the generator's output conditioned on y .
- $D(x|y)$ denotes the discriminator's output for input x conditioned on y .

Pix2Pix GAN: Image-to-Image Translation with Conditional Adversarial Networks

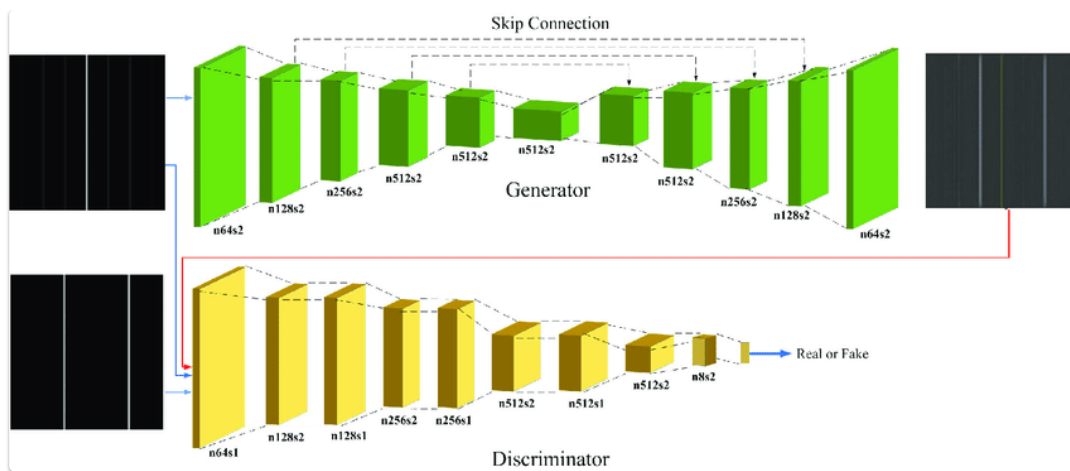
Pix2Pix is a prominent example of a Conditional Generative Adversarial Network (CGAN) specifically designed for **image-to-image translation** tasks. Introduced by Isola et al. in 2017, it demonstrates how a single, general framework can learn to map input images to output images in a diverse range of applications, provided sufficient paired training data.

Core Idea: Learning a General Purpose Image-to-Image Mapping

Unlike traditional image processing algorithms that require hand-crafted rules for specific transformations (e.g., edge detection to photo, semantic labels to photos), Pix2Pix learns this mapping directly from data. It takes an input image and transforms it into a corresponding output image, where the "transformation" is defined by the paired training dataset.

The "condition" (y) in Pix2Pix is not a simple label or vector, but an **entire input image**. The output (x) is the desired translated image.

Architecture of Pix2Pix



Pix2Pix employs a standard GAN setup, but with specific architectural choices for its Generator and Discriminator to handle image inputs and outputs effectively.

1. The Generator (G): U-Net Architecture

The Generator's role is to take an input image (the condition) and produce the corresponding output image. For image-to-image translation, a simple encoder-decoder architecture often struggles to capture fine details, leading to blurry or distorted outputs. Pix2Pix addresses this by using a **U-Net** architecture for its Generator.

- **Encoder-Decoder Structure:** The U-Net still follows the typical encoder-decoder pattern:
 - **Encoder:** Downsamples the input image through a series of convolutional layers, reducing spatial dimensions and extracting hierarchical features.
 - **Decoder:** Upsamples these features to reconstruct the output image with the desired spatial dimensions.
- **Skip Connections:** This is the defining feature of the U-Net. Skip connections directly link layers in the encoder to corresponding layers in the decoder at the same spatial resolution.
 - **Benefit:** These connections allow the decoder to directly access fine-grained details from the encoder's earlier layers that might otherwise be lost during downsampling. This is crucial for generating high-quality, pixel-accurate outputs and for retaining structural information from the input image.
 - **Implementation:** Typically, the feature maps from an encoder layer are concatenated with the feature maps of the corresponding decoder layer before the next convolutional operation in the decoder.
- **Input to Generator:** The input is the source image (e.g., semantic segmentation map, edge map, grayscale image) which serves as the condition. Random noise is often still injected, but its role might be less dominant than in unconditional GANs, as the output is heavily determined by the input image.

2. The Discriminator (D): PatchGAN

Instead of classifying an entire image as real or fake, Pix2Pix uses a **PatchGAN** Discriminator.

- **Local Classification:** A PatchGAN Discriminator classifies overlapping "patches" of the image as real or fake, rather than classifying the entire image globally.
 - **How it works:** The Discriminator's output is not a single scalar, but a 2D grid of values. Each value in this grid corresponds to the Discriminator's assessment of a receptive field (a patch) in the input image.
 - **Benefit:** This design encourages the Generator to produce high-frequency details that are realistic at a local level. It's akin to ensuring that the generated image looks realistic in many small regions, which collectively contributes to a realistic overall image. It also uses fewer parameters than a full-image discriminator, making it faster and more stable to train.
- **Input to Discriminator:**
 - For **real samples**, the Discriminator receives a concatenated pair: `[input_image, real_output_image]`.
 - For **fake samples**, it receives a concatenated pair: `[input_image, generated_output_image]`.
 - The Discriminator learns to identify whether the input image and the corresponding output image are a "plausible" pair (real) or an "implausible" pair (fake).

Loss Functions

Pix2Pix's objective function is a combination of two types of losses: an **adversarial loss** (standard GAN loss) and a **pixel-wise L1 loss** (or L2 loss). This combination is crucial for achieving high-quality results.

1. Adversarial Loss:

This is the standard conditional GAN loss, which aims to make the generated images indistinguishable from real images,

conditioned on the input image.

- **Discriminator Objective (D):** Maximize its ability to classify real pairs as real and fake pairs as fake.

$$L_{GAN}(D) = E_{(x,y) \sim p_{data}(x,y)} [\log D(x, y)] + E_{z \sim p_z(z), y \sim p_{data}(y)} [\log(1 - D(G(z, y), y))]$$

Note: Here, y represents the input image to the Generator, and x is the ground truth output image. $G(z, y)$ is the generated output. The noise z is usually implicit or very small in Pix2Pix, as the output is heavily dependent on the input image y .

- **Generator Objective (G):** Minimize the probability of the Discriminator correctly identifying its output as fake, essentially trying to fool the Discriminator.

$$L_{GAN}(G) = E_{z \sim p_z(z), y \sim p_{data}(y)} [\log D(G(z, y), y)]$$

2. L1 Loss (Reconstruction Loss):

While the adversarial loss encourages realism, it doesn't guarantee that the generated image will be a correct translation of the input. Without this, the Generator might learn to produce realistic-looking images that don't actually correspond to the input (e.g., generate a realistic house when the input was a tree).

- **Objective:** Measures the pixel-wise difference between the generated image $G(z, y)$ and the ground truth real image x .

$$L_{L1}(G) = E_{(x,y) \sim p_{data}(x,y)} [\|x - G(z, y)\|_1]$$

- **Benefit:** The L1 loss (Mean Absolute Error) encourages the Generator to produce images that are structurally similar to the ground truth, helping to prevent mode collapse and ensure the output is a faithful translation of the input. L1 is generally preferred over L2 (Mean Squared Error) in image generation as it encourages less blurring and sharper results.

Full Objective Function: (YOU COULD SKIP THIS IGT)

The final objective for Pix2Pix is a weighted sum of the adversarial loss and the L1 loss:

$$\min_G \max_D L_{GAN}(D) + \lambda L_{L1}(G)$$

where λ (lambda) is a hyperparameter that controls the relative importance of the L1 loss compared to the adversarial loss. The original paper used $\lambda = 100$.

Training Process

The training process for Pix2Pix follows the standard adversarial training loop, adapted for its specific architecture and losses:

- **Step 1: Discriminator Update**
 1. Sample a batch of **real paired data** (y, x) , where y is the input image and x is the corresponding ground truth output image.
 2. Generate a batch of **fake output images** $G(z, y)$ using the current Generator, based on the input images y from the real batch (noise z might be implicit or small).
 3. Feed the real pairs (y, x) to D and compute D 's loss for real samples (aim for high probability).
 4. Feed the fake pairs $(y, G(z, y))$ to D and compute D 's loss for fake samples (aim for low probability).
 5. Combine these losses and update D 's weights using an optimizer (e.g., Adam).
- **Step 2: Generator Update**
 1. Generate a batch of **fake output images** $G(z, y)$ using the current Generator, based on new input images y .
 2. Feed the fake pairs $(y, G(z, y))$ to D .
 3. Compute the **adversarial loss** for G (aim for D to classify them as real).
 4. Compute the **L1 loss** between $G(z, y)$ and the ground truth x from the corresponding input y .
 5. Combine these two losses with the λ weighting and update G 's weights using an optimizer. D 's weights are kept fixed during this step.

This iterative process continues until the Generator is capable of producing high-quality, realistic output images that are faithful translations of the input images, and the Discriminator can no longer reliably distinguish them from real pairs.

CycleGAN: Unpaired Image-to-Image Translation

CycleGAN, introduced by Zhu et al. in 2017, is a groundbreaking extension of GANs that addresses a significant limitation of methods like Pix2Pix: the requirement for **paired training data**. CycleGAN enables image-to-image translation between two different domains (e.g., horses to zebras, summer landscapes to winter landscapes) even when no corresponding input-output image pairs are available.

Core Idea: Cycle Consistency

The central innovation of CycleGAN is the concept of **cycle consistency**. Without paired data, it's impossible to directly enforce that a translated image corresponds accurately to its source. CycleGAN overcomes this by introducing a clever constraint: if you translate an image from domain A to domain B, and then translate the resulting image back from domain B to domain A, you should ideally recover the original image from domain A. This "cycle" acts as a strong regularization, guiding the translation process without direct supervision.

Architecture of CycleGAN

CycleGAN involves **two Generative Adversarial Networks (GANs)**, each consisting of a Generator and a Discriminator. These two GANs operate in a synergistic fashion to enforce cycle consistency.

Let's define our two image domains:

- Domain A: Images X_A (e.g., photos of horses)
- Domain B: Images X_B (e.g., photos of zebras)

1. Two Generators:

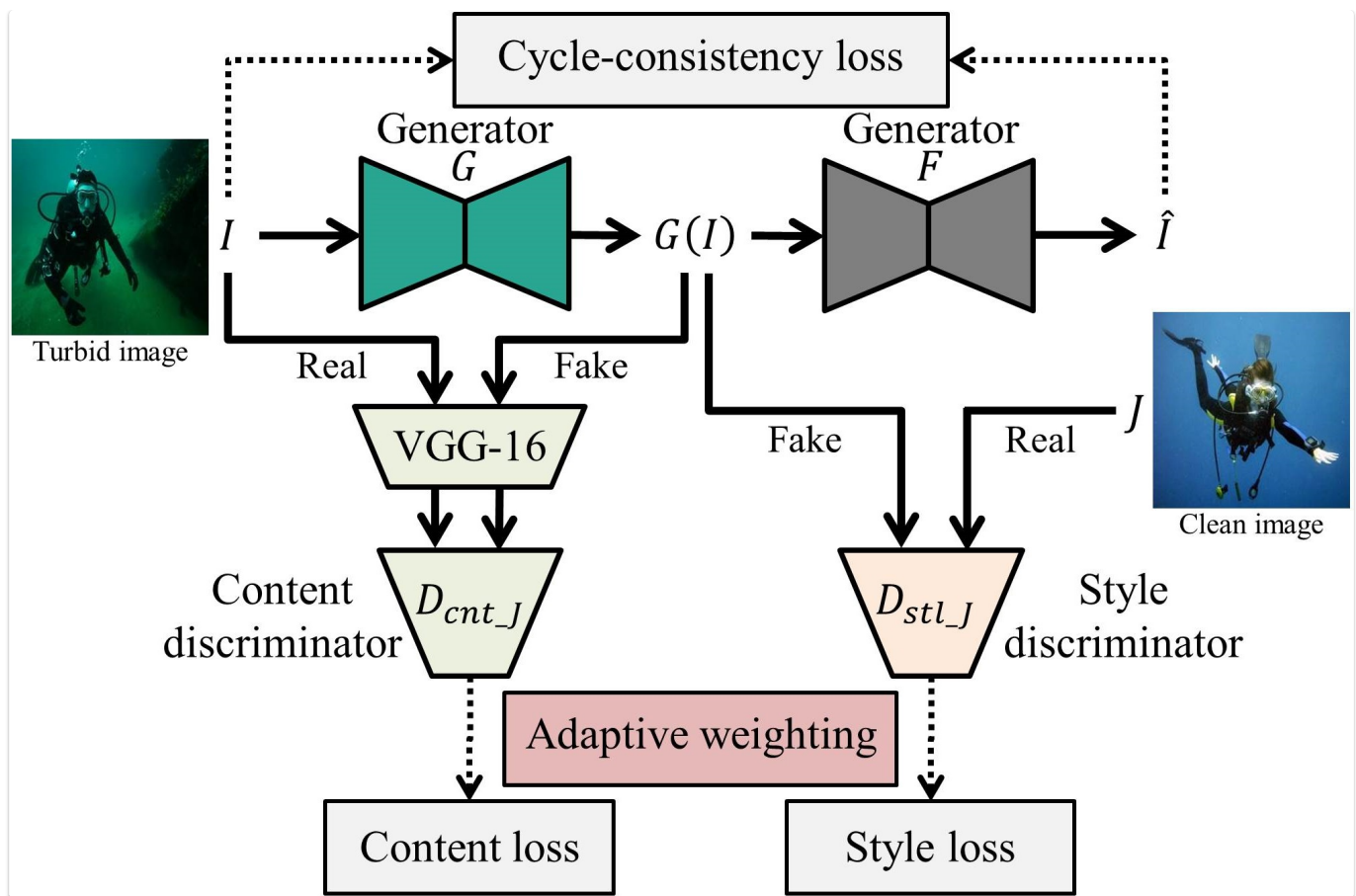
- **Generator $G_{A \rightarrow B}$ (or G)**: Maps images from domain A to domain B.
 - Input: Image from X_A
 - Output: Generated image in domain B, $G_{A \rightarrow B}(x_A)$
- **Generator $G_{B \rightarrow A}$ (or F)**: Maps images from domain B to domain A.
 - Input: Image from X_B
 - Output: Generated image in domain A, $G_{B \rightarrow A}(x_B)$

Both generators typically employ a **U-Net** architecture (similar to Pix2Pix) to leverage skip connections for high-quality image reconstruction and detail preservation.

2. Two Discriminators:

- **Discriminator D_A** : Distinguishes between real images from domain A and fake images generated for domain A.
 - Input: Real image x_A or fake image $G_{B \rightarrow A}(x_B)$
 - Output: Probability of being real in domain A.
- **Discriminator D_B** : Distinguishes between real images from domain B and fake images generated for domain B.
 - Input: Real image x_B or fake image $G_{A \rightarrow B}(x_A)$
 - Output: Probability of being real in domain B.

Both discriminators typically use a **PatchGAN** architecture (similar to Pix2Pix) to encourage realistic textures and high-frequency details at a local patch level.



Example explanation:

After training, Generator G ($A \rightarrow B$) will be able to take a new, unseen turbid underwater image and transform it into a clear, visually appealing image, effectively removing the turbidity, even though it was never trained on paired turbid/clean examples of the same scene. This is achieved by learning the distinct *styles* and *content representations* of turbid vs. clean underwater environments and ensuring that the translation is reversible through the cycle consistency constraint.

Loss Functions: The Cycle Consistency Objective

CycleGAN's full objective function is a combination of three types of losses:

1. Adversarial Loss (2 GANs):

These are standard GAN losses applied to each of the two mapping functions ($G_{A \rightarrow B}$ and $G_{B \rightarrow A}$) and their respective discriminators (D_B and D_A). They ensure that the generated images are realistic within their target domains.

- For $G_{A \rightarrow B}$ and D_B (A to B translation):

$$L_{GAN}(G_{A \rightarrow B}, D_B, X_A, X_B) = E_{x_B \sim p_{data}(x_B)}[\log D_B(x_B)] + E_{x_A \sim p_{data}(x_A)}[\log(1 - D_B(G_{A \rightarrow B}(x_A)))]$$

The Generator $G_{A \rightarrow B}$ tries to minimize this, while D_B tries to maximize it.

- For $G_{B \rightarrow A}$ and D_A (B to A translation):

$$L_{GAN}(G_{B \rightarrow A}, D_A, X_B, X_A) = E_{x_A \sim p_{data}(x_A)}[\log D_A(x_A)] + E_{x_B \sim p_{data}(x_B)}[\log(1 - D_A(G_{B \rightarrow A}(x_B)))]$$

The Generator $G_{B \rightarrow A}$ tries to minimize this, while D_A tries to maximize it.

2. Cycle Consistency Loss:

This is the core of CycleGAN. It enforces that mapping an image from one domain to another and then back to the original domain should reconstruct the original image. This loss is a pixel-wise L1 (Mean Absolute Error) loss.

- Forward Cycle Consistency:** If we take an image from X_A , translate it to X_B , and then translate it back to X_A , it should ideally be identical to the original x_A .

$$x_A \rightarrow G_{A \rightarrow B}(x_A) \rightarrow G_{B \rightarrow A}(G_{A \rightarrow B}(x_A)) \approx x_A$$

$$L_{cyc}(G_{A \rightarrow B}, G_{B \rightarrow A}) = E_{x_A \sim p_{data}(x_A)}[||G_{B \rightarrow A}(G_{A \rightarrow B}(x_A)) - x_A||_1]$$

- Backward Cycle Consistency:** Similarly, if we take an image from X_B , translate it to X_A , and then translate it back to X_B , it should ideally be identical to the original x_B .

$$x_B \rightarrow G_{B \rightarrow A}(x_B) \rightarrow G_{A \rightarrow B}(G_{B \rightarrow A}(x_B)) \approx x_B$$

$$L_{cyc}(G_{A \rightarrow B}, G_{B \rightarrow A}) = E_{x_B \sim p_{data}(x_B)}[||G_{A \rightarrow B}(G_{B \rightarrow A}(x_B)) - x_B||_1]$$

The total cycle consistency loss is the sum of the forward and backward components.

3. Identity Loss (Optional but Recommended):

This loss encourages the generators to preserve color composition between the input and output images during translation, especially when the translation task doesn't involve significant color shifts (e.g., changing objects but keeping the background color). It's typically applied as an L1 loss.

- **For $G_{A \rightarrow B}$:** If a real image from domain B is fed into $G_{A \rightarrow B}$, it should ideally remain unchanged.

$$L_{id}(G_{A \rightarrow B}) = E_{x_B \sim p_{data}(x_B)}[||G_{A \rightarrow B}(x_B) - x_B||_1]$$

- **For $G_{B \rightarrow A}$:** If a real image from domain A is fed into $G_{B \rightarrow A}$, it should ideally remain unchanged.

$$L_{id}(G_{B \rightarrow A}) = E_{x_A \sim p_{data}(x_A)}[||G_{B \rightarrow A}(x_A) - x_A||_1]$$

The total identity loss is the sum of these two.

Full Objective Function:

The full objective for CycleGAN is the weighted sum of all these losses. The generators aim to minimize this objective, while the discriminators aim to maximize it:

$$\min_{G_{A \rightarrow B}, G_{B \rightarrow A}} \max_{D_A, D_B} L(G_{A \rightarrow B}, G_{B \rightarrow A}, D_A, D_B)$$

where:

$$L(G_{A \rightarrow B}, G_{B \rightarrow A}, D_A, D_B) = L_{GAN}(G_{A \rightarrow B}, D_B, X_A, X_B) + L_{GAN}(G_{B \rightarrow A}, D_A, X_B, X_A) + \lambda_{cyc} L_{cyc}(G_{A \rightarrow B}, G_{B \rightarrow A}) + \lambda_{id} L_{id}$$

- λ_{cyc} : Weight for the cycle consistency loss (e.g., 10 in the original paper). This is critical for preventing the generators from producing arbitrary outputs.
- λ_{id} : Weight for the identity loss (e.g., $0.5 * \lambda_{cyc}$ in the original paper).

Training Process

CycleGAN's training is more complex than a standard GAN due to the multiple interacting components. It involves iterating through the following steps:

1. Discriminator Updates:

- **Update D_A :**
 - Sample real images x_A from domain A.
 - Generate fake images $G_{B \rightarrow A}(x_B)$ from domain B using $G_{B \rightarrow A}$.
 - Train D_A to classify x_A as real and $G_{B \rightarrow A}(x_B)$ as fake using its L_{GAN} component.
- **Update D_B :**
 - Sample real images x_B from domain B.
 - Generate fake images $G_{A \rightarrow B}(x_A)$ from domain A using $G_{A \rightarrow B}$.
 - Train D_B to classify x_B as real and $G_{A \rightarrow B}(x_A)$ as fake using its L_{GAN} component.

2. Generator Updates:

- **Update $G_{A \rightarrow B}$ and $G_{B \rightarrow A}$ simultaneously:**
 - Compute the adversarial loss for $G_{A \rightarrow B}$ (based on D_B 's output for $G_{A \rightarrow B}(x_A)$).
 - Compute the adversarial loss for $G_{B \rightarrow A}$ (based on D_A 's output for $G_{B \rightarrow A}(x_B)$).
 - Compute the forward cycle consistency loss ($||G_{B \rightarrow A}(G_{A \rightarrow B}(x_A)) - x_A||_1$).
 - Compute the backward cycle consistency loss ($||G_{A \rightarrow B}(G_{B \rightarrow A}(x_B)) - x_B||_1$).
 - Compute the identity loss for $G_{A \rightarrow B}$ ($||G_{A \rightarrow B}(x_B) - x_B||_1$).
 - Compute the identity loss for $G_{B \rightarrow A}$ ($||G_{B \rightarrow A}(x_A) - x_A||_1$).
 - Sum all these losses (with their respective λ weights) and update the weights of **both** $G_{A \rightarrow B}$ and $G_{B \rightarrow A}$. The Discriminators' weights are frozen during this step.

This iterative process, with its sophisticated combination of adversarial and cycle consistency losses, allows CycleGAN to learn complex image transformations without the prohibitive requirement of paired training data. It implicitly discovers the mapping between domains

by ensuring that images can be translated and then translated back to their origin.

StyleGAN: High-Quality and Controllable Image Synthesis

StyleGAN (and its subsequent versions StyleGAN2 and StyleGAN3) represents a significant leap in the field of generative models, particularly for the synthesis of highly realistic and controllable images, especially human faces. Developed by NVIDIA, StyleGAN builds upon the foundational principles of GANs by re-architecting the Generator to leverage "styles" at different levels of detail, leading to unprecedented control over the generated image's features.

Core Idea: Style-Based Generator Architecture

The central innovation of StyleGAN is its **style-based generator architecture**. Instead of feeding the latent code (noise) directly into the first layer of the generator, StyleGAN introduces it through multiple "style" inputs at different layers of the network. This allows for disentangled control over various visual features, from high-level attributes (e.g., pose, face shape) to fine details (e.g., hair color, freckles).

Key Components of the StyleGAN Generator:

1. Mapping Network (f):

- **Purpose:** This small, 8-layer MLP (Multi-Layer Perceptron) transforms the initial, randomly sampled latent code z (typically from a normal distribution) into an intermediate latent space w .
- **Benefit:** The intermediate latent space w is designed to be more "disentangled" than the original z . This means that changes in individual dimensions of w are more likely to correspond to changes in single, interpretable features of the generated image, without affecting other features. This disentanglement is crucial for precise control.
- **Output:** The mapping network produces multiple w vectors, one for each "style" input in the synthesis network.

2. Synthesis Network (g):

- **Purpose:** This is the core image generation part of the network. It's a progressive growing-like architecture (similar to PGGAN) that builds the image layer by layer, from low resolution to high resolution.
- **Constant Input:** Unlike traditional GANs that start with a noise vector as spatial input, the synthesis network begins with a learned **constant $4 \times 4 \times 512$ tensor**. This means the initial spatial content is fixed, and all variations come from the styles.
- **Adaptive Instance Normalization (AdaIN) / Style Modulation:** At each convolutional layer of the synthesis network, the w vector (a "style") is used to **modulate** the features via AdaIN. AdaIN normalizes the feature maps (removes mean and variance per channel) and then applies learned scale and bias factors derived from the style vector.
 - **Effect:** This modulation at each resolution effectively injects style information, controlling attributes specific to that resolution level. For example, early layers' styles might control global features like pose or general facial structure, while later layers' styles might control finer details like skin texture or hair strands.
- **Per-Pixel Noise:** Random Gaussian noise is added to the output of each convolutional layer (before AdaIN).
 - **Benefit:** This noise allows the network to generate stochastic details (e.g., hair strands, freckles, wrinkles) that are specific to each generated image, without being explicitly encoded in the latent space. This adds realism and prevents "plastic" looking outputs.
 - **Implementation:** The noise is added to feature maps at a specific resolution and then scaled by a learned per-channel weight.

3. Discriminator:

- The Discriminator typically uses a standard convolutional architecture (similar to DCGAN or PGGAN's discriminator), trained to distinguish between real and fake images. It receives the generated image from the Synthesis Network and a real image from the dataset.

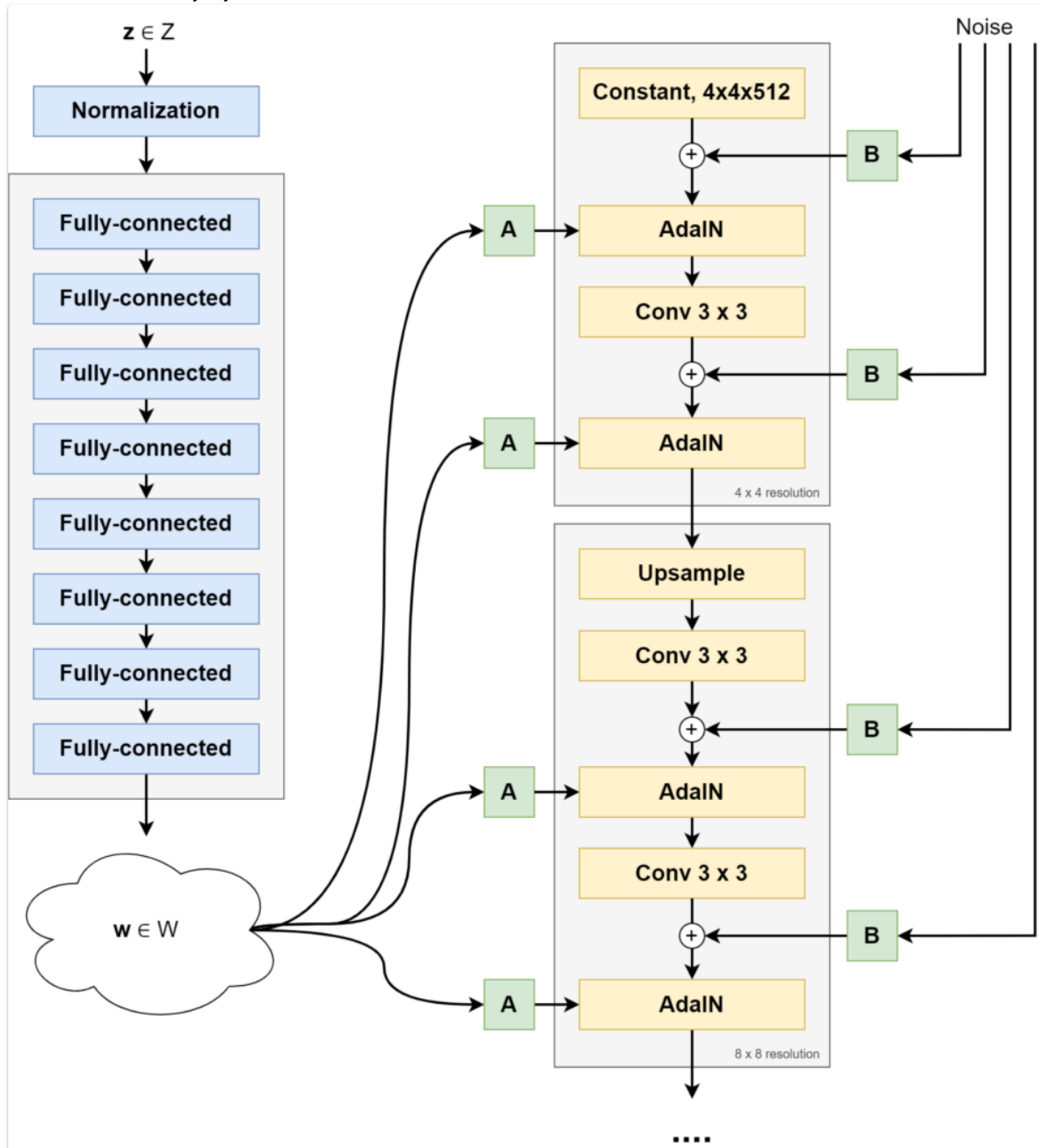
How StyleGAN Works: Disentangled Control

The power of StyleGAN comes from the way the style (modulated w) is injected at different levels:

- **Coarse Styles (Early Layers):** When w modulates features in the early, low-resolution layers of the synthesis network, it affects high-level attributes that span the entire image, such as:
 - Gender
 - Age
 - Pose
 - Hair style

- Overall face shape
- **Middle Styles (Middle Layers):** Styles injected at intermediate resolutions control more localized features like:
 - Eye color
 - Hair color
 - Skin tone
 - Facial expression
- **Fine Styles (Later Layers):** Styles affecting the high-resolution layers control subtle, fine-grained details such as:
 - Freckles
 - Pores
 - Specific hair strands
 - Minor lighting variations

This layered injection allows for **disentanglement**: you can manipulate styles at one resolution level without significantly affecting features controlled by styles at other resolution levels.



Loss Functions

| This loss function is fucked up and they can eat shit.

Simplified StyleGAN Loss

Generator Loss (G):

$$\mathcal{L}_G = -\mathbb{E}_{z \sim P(z)} [\log D(G(z))] + \lambda_{\text{PL}} \cdot \mathcal{L}_{\text{PL}}$$

Discriminator Loss (D):

$$\mathcal{L}_D = -\mathbb{E}_{x \sim P_{\text{real}}} [\log D(x)] - \mathbb{E}_{z \sim P(z)} [\log(1 - D(G(z)))] + \lambda_{\text{R1}} \cdot \mathcal{L}_{\text{R1}}$$

Where:

- $G(z)$ = generated image from latent code z
- $D(x)$ = discriminator output (probability that x is real)
- $\mathcal{L}_{\text{PL}}$ = path length regularization
- $\mathcal{L}_{\text{R1}}$ = R1 gradient penalty (only on real images)
- $\lambda_{\text{PL}}, \lambda_{\text{R1}}$ = weights for regularization terms