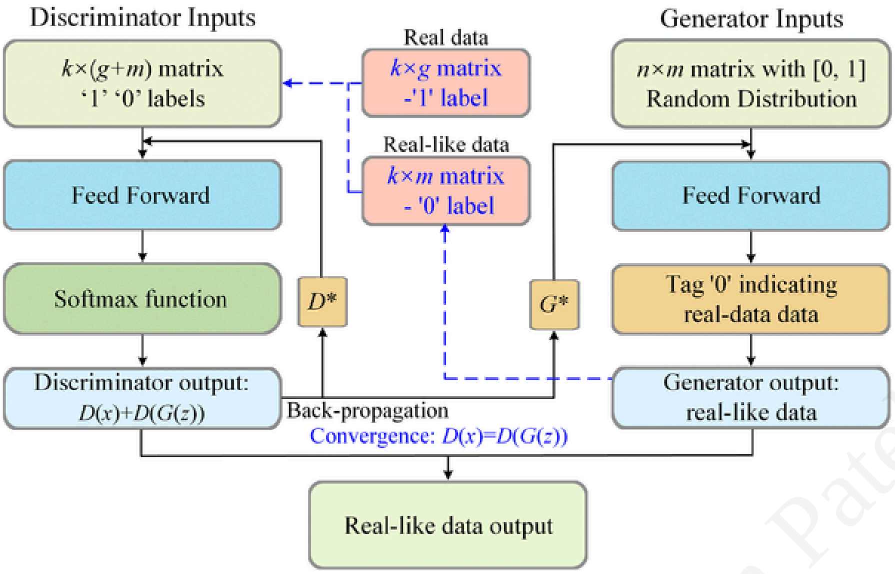
Practical 8

To Understand & Implement Generative Adversarial Network (GAN) Model



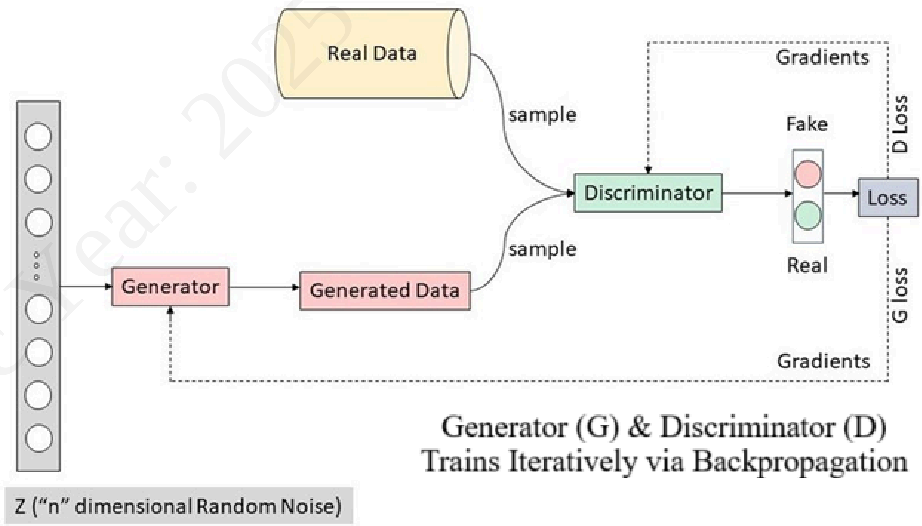
Theory:

A generative adversarial network (GAN) is an artificial intelligence framework that uses two competing neural networks—a generator and a discriminator—to create new data samples that are indistinguishable from a real training dataset.

The generator learns to produce data, while the discriminator learns to differentiate between real and fake data.

Through this adversarial training process, the generator becomes increasingly proficient at creating realistic synthetic data, such as images of non-existent people, realistic animal images, or new fashion designs.

How GANs Work



- Generator: This neural network takes random noise as input and transforms it into a data sample, such as an image.
- Discriminator: This network receives both real data samples from the training set and fake data samples from the generator. Its role is to classify whether a sample is real or fake.
- Adversarial Training: The generator and discriminator are trained against each other.

> The generator aims to fool the discriminator by producing more convincing fake data.

> The discriminator aims to become better at identifying the fake samples.

- Goal: The ultimate goal is for the generator to produce data so realistic that the discriminator can no longer tell the difference between the real and fake samples.

Applications of GANs

- Image Generation: Creating photorealistic images of people, animals, and other objects that do not exist. Video Synthesis: Generating realistic video content.
- 3D Model Creation: Synthesizing detailed 3D models for applications like video games and virtual reality.
- Data Augmentation: Generating synthetic data to supplement existing datasets, which is particularly useful for training deep learning models when real data is scarce.
- Image-to-Image Translation: Converting an image from one domain to another (e.g., turning a sketch into a photorealistic image).
- Text-to-Image Synthesis: Creating images based on textual descriptions.

Types of Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) come in various forms, each designed to address specific challenges or improve generation quality. Below are the major types and their core ideas.

---

1. Vanilla GAN

Overview:
Vanilla GAN is the most basic form of GAN. It consists of:

- A **Generator** and a **Discriminator**, both implemented using *Multi-Layer Perceptrons (MLPs)*.
- Optimization performed using *Stochastic Gradient Descent (SGD)*.

Characteristics:

- The generator creates synthetic data.
- The discriminator attempts to distinguish between real and fake samples.

Limitations:

- ⚠️ *Mode Collapse*: The generator may produce only a few variations of samples.
- ⚠️ *Unstable Training*: The learning process of generator and discriminator can oscillate.

---

2. Conditional GAN (CGAN)

Concept:
CGAN introduces **conditioning** — an additional parameter that guides what type of data the generator should produce.

How it Works:

- A conditional variable **y** (e.g., label or class information) is fed into both the generator and discriminator.
- The generator learns to produce data corresponding to a specific label.
- The discriminator uses this label to improve its judgment.

Example:
Instead of generating any random image, CGAN can generate:

- A *dog* image if the label is “dog”
- A *cat* image if the label is “cat”

---

3. Deep Convolutional GAN (DCGAN)

Concept:
DCGAN replaces fully connected layers with **Convolutional Neural Networks (CNNs)** for more realistic image synthesis.

Key Features:

- Uses **convolutional** and **transposed convolutional** layers.
- Removes *max pooling*, replacing it with *strided convolutions*.
- Removes *fully connected layers* to preserve spatial information.

Benefits:

- Generates highly realistic images.
- Trains more stably compared to Vanilla GANs.

---

4. Laplacian Pyramid GAN (LAPGAN)

Concept:
LAPGAN uses a **multi-resolution (pyramid) approach** to create highly detailed images.

How it Works:

1. Multiple **generator–discriminator pairs** operate at different pyramid levels.
2. Images are **downsampled** and **upsampled** progressively.
3. Each stage adds more fine-grained details using conditional GANs.

Advantages:

- Produces high-resolution, photorealistic outputs.
- Gradual refinement reduces noise and enhances clarity.

---

5. Super-Resolution GAN (SRGAN)

Concept:
SRGAN focuses on **image super-resolution** — transforming low-resolution images into high-resolution ones.

Mechanism:

- Combines a deep neural network with **adversarial loss**.
- The generator learns to upscale and enhance image details.
- The discriminator ensures realism and sharpness.

Applications:

- Improving old or blurry photographs.
- Enhancing satellite or medical images.

---

6. Cycle GAN

Concept:
CycleGAN enables **image-to-image translation without paired data**.
It learns to translate between two domains (e.g., horses ↔ zebras, summer ↔ winter).

How it Works:

- Two generators and two discriminators are used:
  - One generator translates **Domain A → Domain B**.
  - The other does **Domain B → Domain A**.
- A **cycle-consistency loss** ensures that translating back gives the original image.

Advantages:

- Does **not require paired training data**.
- Useful for style transfer, domain adaptation, and artistic transformation.

🧩 Summary Table

Type	Key Idea	Architecture	Main Application
Vanilla GAN	Basic adversarial training	MLP-based	General image synthesis
CGAN	Conditional data generation	MLP + conditioning	Class-controlled generation
DCGAN	CNN-based GAN	CNNs & ConvTranspose	Realistic image generation
LAPGAN	Multi-scale pyramid	Hierarchical GANs	High-resolution image generation
SRGAN	Super-resolution	CNN + GAN loss	Image upscaling
CycleGAN	Unpaired translation	Dual GANs + Cycle loss	Image-to-image translation

📘 *GANs continue to evolve, with newer variants such as StyleGAN, BigGAN, and Diffusion Models pushing the boundaries of generative modeling.*

- Style-based GANs (StyleGAN family) excel in fine control over features like facial attributes, textures, and lighting.
- Attention-based GANs (SAGAN, AttnGAN) allow capturing long-range dependencies and improve semantic alignment.
- WGAN variants (WGAN-GP, etc.) improve stability and gradient flow — critical for high-res data (like your 100×100 images).
- CycleGAN and Pix2Pix dominate image-to-image translation, while SRGAN focuses on super-resolution.
- ProGAN and BigGAN are designed for large-scale, high-fidelity synthesis.

GAN Type	Year	Core Idea / Improvement	Key Feature	Architecture / Loss Function	Use Cases
Vanilla GAN	2014	Original GAN framework	Minimax adversarial loss	MLP (fully connected) for both generator & discriminator	Basic image generation (MNIST, etc.)
DCGAN (Deep Convolutional GAN)	2015	Introduced CNNs in GANs	Convolution + BatchNorm for stability	Conv/Deconv layers, LeakyReLU	Image generation, feature learning
CGAN (Conditional GAN)	2014	Added class labels as conditions	Conditional generation	Input condition concatenated to G & D	Class-specific image generation
InfoGAN	2016	Disentangled latent representations	Mutual information loss	Extra latent code vector	Controllable generation (e.g. digit rotation)
LSGAN (Least Squares GAN)	2017	Modified loss for stability	Uses L2 loss instead of log loss	Least-squares loss	Reduces vanishing gradient, more stable training
WGAN (Wasserstein GAN)	2017	Introduced Earth Mover's (Wasserstein) distance	Continuous, smoother loss	Weight clipping	Stable training, better convergence
WGAN-GP (WGAN with Gradient Penalty)	2017	Improved over WGAN	Gradient penalty replaces weight clipping	Adds Lipschitz constraint	Stable high-quality image synthesis
EBGAN (Energy-Based GAN)	2016	Replaced discriminator with autoencoder	Energy-based loss	Autoencoder reconstruction energy	Anomaly detection, feature learning
BEGAN (Boundary Equilibrium GAN)	2017	Balanced generator/discriminator learning	Equilibrium hyperparameter *k*	Autoencoder-based discriminator	Human faces, natural images
CycleGAN	2017	Image-to-image translation without paired data	Cycle consistency loss	Two generators + two discriminators	Style transfer, domain adaptation
Pix2Pix	2016	Image translation with paired datasets	Conditional GAN loss	Encoder-decoder generator (U-Net)	Image-to-image translation (sketch→photo)
StyleGAN	2018	Style-based generation	Style vectors at each layer	Progressive growing + AdaIN	Photorealistic faces (FFHQ)
StyleGAN2	2019	Improved over StyleGAN	Removed "blob" artifacts	Weight demodulation	High-resolution, natural-looking images
StyleGAN3	2021	Solved aliasing issues	Continuous feature mapping	Alias-free convolution	Realistic videos and motion generation
ProGAN (Progressive Growing GAN)	2018	Gradually increases image resolution	Layer-wise training	Progressive upsampling	High-res image generation
BigGAN	2018	Scalable GAN with large batch size	Spectral normalization	Deep residual architecture	Large-scale class-conditional images
SRGAN (Super-Resolution GAN)	2017	GAN for super-resolution	Perceptual loss + content loss	Residual generator + VGG loss	Image super-resolution (4×, 8×)
SAGAN (Self-Attention GAN)	2019	Added self-attention to capture global context	Self-Attention layer	Non-local operations	Object coherence, fine details

GAN Type	Year	Core Idea / Improvement	Key Feature	Architecture / Loss Function	Use Cases
AttnGAN	2018	Text-to-image synthesis	Attention mechanism between text and image	Multi-stage generator	Text-guided image generation
Pix2PixHD	2018	High-resolution version of Pix2Pix	Multi-scale generators/discriminators	Coarse-to-fine structure	Semantic map → photorealistic image
SPA-GAN (Style-Preserving Attention GAN)	2020	Improved texture and color transfer	Style + attention fusion	Hybrid loss (content + style)	Artistic style transfer
StyleGAN-T (Text-based)	2023	Language-guided StyleGAN	CLIP latent alignment	CLIP + StyleGAN fusion	Text-conditioned image generation
Diffusion-GAN (Hybrid)	2022	Combines GAN and diffusion advantages	Dual training signal	Discriminator-guided diffusion	High-quality & stable image synthesis

GAN Losses & Architectures

Quick Legend / Variables

- `x` : real data sample (from data distribution `p_data`)
- `z` : latent noise vector (sampled from prior `p_z` , e.g. normal N(0,I))
- `y` : condition / label (for conditional GANs)
- `G(z)` : generator output (fake sample produced from noise `z`)
- `D(·)` : discriminator output (probability or logit)
- `f_w(·)` : critic function in WGAN (real-valued score)
- `E[...]` : expectation (in practice empirical average over batch)
- `I(a; b)` : mutual information between `a` and `b`
- `λ, λ_cyc, λ_adv` : scalar weights for auxiliary losses

Table of Common GAN Types (Losses)

GAN Type	Training Objective (Loss, plain text)	Short Notes / Architecture
Vanilla GAN (original)	$\min_G \max_D V(D,G) = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$ Practical generator loss: $L_G = - E_z[\log D(G(z))]$	Simple MLP or CNN; uses binary cross-entropy. Susceptible to instability and mode collapse.
Conditional GAN (cGAN)	$\min_G \max_D V(D,G) = E_{x,y}[\log D(x \mid y)] + E_{z,y}[\log(1 - D(G(z\|y) \mid y))]$	Generator and discriminator both receive condition `y` ; useful for class-conditioned outputs.
LSGAN (Least Squares GAN)	$L_D = 1/2 * E_x[(D(x)-b)^2] + 1/2 * E_z[(D(G(z)) - a)^2]$ $L_G = 1/2 * E_z[(D(G(z)) - c)^2]$ (typical a=0, b=1, c=1)	Uses L^2 loss to reduce vanishing gradients and stabilize training.
WGAN (Wasserstein GAN)	$L_D = - E_x[f_w(x)] + E_z[f_w(G(z))]$ $L_G = - E_z[f_w(G(z))]$ (critic `f_w` should be 1-Lipschitz)	Replaces discriminator with a critic (outputs real-valued scores). Use weight clipping or gradient penalty. Stable training.
WGAN-GP	$L_D = -E_x[f_w(x)] + E_z[f_w(G(z))] + \lambda * E_{\hat{x}}[(\| \nabla_{\hat{x}} f_w(\hat{x}) \|_2 - 1)^2]$	Gradient penalty enforces Lipschitz condition; standard for stable training.
InfoGAN	$\min_G \max_D V(D,G) - \lambda * I(c; G(z,c))$	c is structured latent code (discrete/continuous) — encourages disentangled, controllable factors.
Pix2Pix	$L_{cGAN} = E_{x,y}[\log D(x,y)] + E_{x,z}[\log(1 - D(x, G(x,z)))]$ $L_G = L_{cGAN} + \lambda * E_{x,y,z}[\| y - G(x,z) \|_1]$	Paired examples; generator is U-Net, discriminator often PatchGAN.
CycleGAN	Two GAN losses + cycle-consistency: $L = L_{GAN}(G,D_B,A,B) + L_{GAN}(F,D_A,B,A) + \lambda_{cyc} * (E_x[\| F(G(x)) - x \|_1] + E_y[\| G(F(y)) - y \|_1])$	Two generators (A→B and B→A) + two discriminators; cycle loss enforces mapping consistency without paired data.
SRGAN	$L_G = L_{content} + \lambda_{adv} * L_{adv}$ $L_{content} = \| \varphi(y) - \varphi(G(x)) \|_2^2$ (perceptual loss)	Generator uses residual blocks; discriminator ensures realism; perceptual loss preserves image semantics and sharpness.
StyleGAN / StyleGAN2 / StyleGAN3	$\min_G \max_D L_{adv}(G,D)$ + regularizers; mapping z → w, per-layer style via affine/AdaIN or weight demodulation	Style-based generator; supports style-mixing, path-length regularization, alias-free operations. State-of-the-art photorealism.
BigGAN	Class-conditional adversarial objectives (often hinge loss). Example hinge D loss: $L_D = E[\max(0,1-D(x,y))] + E[\max(0,1+D(G(z,y),y))]$	Large-scale, class-conditional model; strong regularization; excellent ImageNet results.

SAGAN	Standard adversarial loss + self-attention layers inside G/D	Self-attention modules capture long-range dependencies; improves object coherence.

Short Notes on Variables

- `E[...]` means expectation (compute mean over mini-batch).
- `G(z)` = generator output given noise `z`.
- `D(x)` usually outputs probability [0,1] (vanilla GAN) or real-valued score (WGAN).
- $f_w(x)$ denotes critic output in WGAN (no sigmoid).
- Constants like `a, b, c` (LSGAN) or `λ` are hyperparameters.
- Perceptual losses use feature extractor `ϕ(·)`, e.g. pretrained VGG intermediate activations.

In [1]:
```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import os, glob, imageio
from IPython import display
import tensorflow_docs.vis.embed as embed
```

In [2]:
```
# ============================================================
# 0. Weights Initialization
# ============================================================

gen_losses, disc_losses = [], []
```

choosing the **latent dimension (latent_dim)** in GANs is not arbitrary, but there's no *hard formula*. Instead, it's about **balancing expressive power vs training stability**.

---

1. Role of Latent Dim

- The **latent vector (z)** is like a compressed code that the generator expands into an image.
- Think of it as the **DNA of an image**:
  - Small latent_dim → limited "genetic material" → less diversity in images.
  - Large latent_dim → too much "freedom" → generator may struggle to map all noise → unstable training.

---

2. Rules of Thumb

#(a) **Start Simple**

- For **small/simple datasets** (MNIST digits, simple icons): `latent_dim = 50-100` is enough.
- For **medium complexity** (faces, anime characters, fashion): `latent_dim = 100-200` is common.
- For **high-resolution / very diverse images** (ImageNet, realistic faces): `latent_dim = 256-512` (or even higher in StyleGAN).

(b) **Relation to Image Size**

- Roughly: the **more pixels / details**, the more latent dimensions you need.
- Example:
  - MNIST (28×28 grayscale) → 50–100
  - Anime (64×64 or 100×100 RGB) → 100–200
  - CelebA-HQ (256×256) → 256–512

But note: **image resolution is not the only factor** — dataset diversity matters a lot too. E.g., 100×100 images of just 1 anime character need less latent_dim than 100×100 images of 1000 unique characters.

(c) **Batch Size & Latent Dim**

- **Independent choices** → latent_dim doesn't depend on batch size.
- Batch size controls **how many noise vectors per training step**, not the structure of noise.

---

3. Practical Approach

1. Start with **latent_dim = 100** (default for most DCGANs).
2. Train for a few epochs.
3. If:

- Generated images look **too similar** → increase latent_dim.
- Training is **unstable / generator collapses** → decrease latent_dim.

---

4. Example

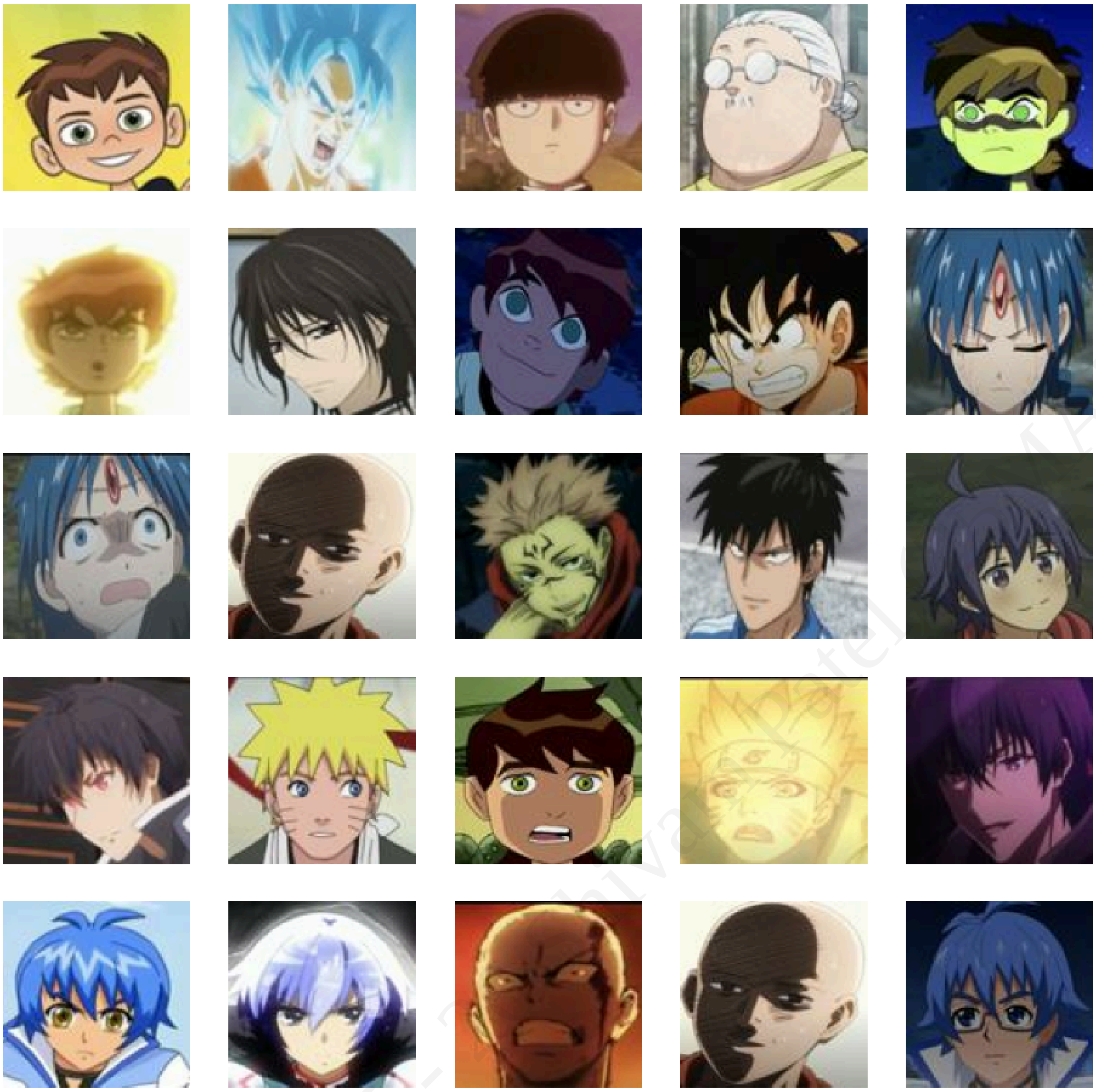
```
LATENT_DIM = 100   # good starting point
noise = tf.random.normal([32, LATENT_DIM])  # batch_size = 32
fake_images = generator(noise)  # (32, 64, 64, 3) if RGB anime dataset
```

So the **decision comes from dataset complexity + image resolution**.

In [3]:
```
# ============================================================
# 1. Load Anime Character Dataset
# ============================================================
img_size = 100
BATCH_SIZE = 16
batch_size = BATCH_SIZE # as data is small
LATENT_DIM = 200#64 # smaller latent space for faster training

# Load dataset from folder
dataset = tf.keras.utils.image_dataset_from_directory(
    "processed",                # path to folder containing images
    label_mode=None,         # no labels needed
    image_size=(img_size, img_size),
    batch_size=batch_size,
    shuffle=True
)

# Normalize images to [-1, 1]
dataset = dataset.map(lambda x: (tf.cast(x, tf.float32) - 127.5) / 127.5)

# Optional: prefetch for performance
dataset = dataset.prefetch(buffer_size=tf.data.AUTOTUNE)
```

```
Found 580 files.
```

In [4]:
```
def plot_images(images):
    plt.figure(figsize=(10,10))
    for i in range(25):
        plt.subplot(5,5,i+1)
        plt.imshow((images[i] * 127.5 + 127.5).numpy().astype("int32"))
        plt.axis('off')
    plt.show()
for image_batch in dataset.take(1):
    images = tf.concat([image_batch for image_batch in dataset.take(8)], axis=0)  # 4*8=32 images
    plot_images(images)
    print(image_batch.shape)
```



```
(16, 100, 100, 3)
```

While GAN architectures are somewhat flexible, there are **some practical rules and guidelines** people follow to define **generator and discriminator architectures** effectively.

---

1. Generator Rules

The generator's job is to take a **latent vector (noise)** and generate realistic images.

Guidelines:

1. **Start with Dense + Reshape**
   - Map latent vector (`z`) to a small "spatial feature map"
   - Example: `latent_dim=100` → Dense → reshape `(8×8×256)` for 64×64 images
2. **Use Conv2DTranspose / UpSampling2D**
   - Gradually increase spatial size: 8→16→32→64
   - Use `stride=2` for doubling dimensions
3. **Batch Normalization**
   - After each layer except the output, stabilizes training and prevents mode collapse
4. **Activation Functions**
   - **Hidden layers:** ReLU or LeakyReLU
   - **Output layer:** Tanh (if images are scaled to [-1,1]) or Sigmoid (if [0,1])
5. **Final output size**
   - Use the formula: [Initial size ≈ final image size / (2 ^ number of upsampling layers)]

6. **Channels**
    - Output channels = number of image channels (1 for grayscale, 3 for RGB)

---

2. Discriminator Rules

The discriminator's job is to classify **real vs fake** images.

Guidelines:

1. **Input = image size**
    - Shape must match the generator output
2. **Use Conv2D**
    - Progressive downsampling: stride=2 halves the spatial dimensions each layer
3. **LeakyReLU**
    - Use LeakyReLU instead of ReLU to avoid dead neurons
4. **Dropout**
    - Helps regularize discriminator and prevent overfitting
5. **Flatten and Dense(1)**
    - At the end, flatten features and output single value for real/fake
6. **No BatchNorm in first layer**
    - Optional in deeper layers, but avoid in first layer (helps stabilize training)

---

3. Other Practical Key Points

- **Generator → Discriminator "mirror"**
    - Often the generator and discriminator are roughly symmetric in layer count, but mirrored (upsampling vs downsampling)
- **Number of filters**
    - Generator: fewer filters in early layers, more filters in later layers
    - Discriminator: more filters in early layers, gradually fewer toward the end
- **Start simple**
    - For 64×64: 3–4 Conv layers in both networks are enough
    - For 100×100 or higher: increase layers or filters to capture complexity
- **Use rules of thumb for output size**
    - For generator: initial feature map = final image size ÷ 2^n
    - For discriminator: output = 1 scalar per image

✅ **Summary:**

Network	Key Points
Generator	Dense+Reshape → Conv2DTranspose → BatchNorm → ReLU → Output Tanh
Discriminator	Conv2D → LeakyReLU → Dropout → Flatten → Dense(1)
Tips	Mirror architectures, progressive up/downsampling, adjust filters, avoid BN in first layer of discriminator

---

This is about **how the spatial dimensions are computed in the generator**.

How to decide `Reshape`

In DCGAN, the generator starts from a **dense layer** and reshapes it into a small "image" which is then **upsampled** via `Conv2DTranspose` layers.

Upsampling formula for Conv2DTranspose with stride 2, padding "same":

[Output size = Input size × stride = h × s]

Example: 64×64 output vs 100×100 output

Original 64×64 generator (MNIST/anime 64×64):

- Dense layer → reshape `(8,8,256)`
- Conv2DTranspose stride=2 → 8→16
- Conv2DTranspose stride=2 → 16→32
- Conv2DTranspose stride=2 → 32→64

So `8×8` is enough for 64×64 output.

New 100×100 generator:

- Conv2DTranspose with stride 2 multiplies the size:
  - Suppose we keep 3 layers with stride=2 each:
    - 12 → 24 → 48 → 96
- Close to 100, then final layer can **pad or crop** to 100×100.

If we had kept `8×8`:

- 8 → 16 → 32 → 64 → too small for 100×100

Hence we increase the **starting feature map** to `12×12` so that after 3 transposed conv layers we get ~96×96, and last layer adjusts to 100×100.

Rule of thumb:

$$\text{Initial size} \approx \frac{\text{final image size}}{2^{\text{num upsampling layers}}}$$

- For 64×64, `64/2^3 = 8` → initial `8×8`
- For 100×100, `100/2^3 ≈ 12` → initial `12×12`

In [5]:
```
# ============================================================
# 2. Build Generator
# ============================================================
def build_generator(latent_dim=LATENT_DIM):
    model = tf.keras.Sequential([
        layers.InputLayer(shape=(latent_dim,)),
        layers.Dense(25*25*256, use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Reshape((25, 25, 256)),

        layers.Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),  # (50, 50, 128)

        layers.Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),  # (100, 100, 64)

        layers.Conv2DTranspose(3, (3,3), strides=(1,1), padding='same', use_bias=False, activation='tanh') # if st
        #layers.Lambda(lambda x: tf.image.resize(x, [100, 100]))
    ])
    return model
```

In [6]:
```
# ============================================================
# 3. Build Discriminator
# ============================================================
def build_discriminator(input_shape=(100, 100, 3)):
    model = tf.keras.Sequential([
        layers.InputLayer(shape=input_shape),
        # 100 -> 50
        layers.Conv2D(64, (3,3), strides=(2,2), padding="same"),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
```

```
        # 50 -> 25
        layers.Conv2D(128, (3,3), strides=(2,2), padding="same"),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
        # 25 -> 12
        layers.Conv2D(256, (3,3), strides=(2,2), padding='same'),
        layers.LeakyReLU(0.2),
        layers.Dropout(0.3),
        # 12 -> 6
        layers.Conv2D(512, (3,3), strides=(2,2), padding='same'),
        layers.LeakyReLU(0.2),
        layers.Dropout(0.3),
         # Flatten → Binary output
        layers.Flatten(),
        layers.Dense(1)
    ])
    return model
```

In [15]:

```
# ============================================================
# 4. Loss and Optimizers
# ============================================================
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator = build_generator()
discriminator = build_discriminator()

print("Generator Summary:")
generator.summary()
print("\nDiscriminator Summary:")
discriminator.summary()

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Generator Summary:

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 160000)	32,000,000
batch_normalization_3 (BatchNormalization)	(None, 160000)	640,000
leaky_re_lu_7 (LeakyReLU)	(None, 160000)	0
reshape_1 (Reshape)	(None, 25, 25, 256)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 50, 50, 128)	294,912
batch_normalization_4 (BatchNormalization)	(None, 50, 50, 128)	512
leaky_re_lu_8 (LeakyReLU)	(None, 50, 50, 128)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 100, 100, 64)	73,728
batch_normalization_5 (BatchNormalization)	(None, 100, 100, 64)	256
leaky_re_lu_9 (LeakyReLU)	(None, 100, 100, 64)	0
conv2d_transpose_5 (Conv2DTranspose)	(None, 100, 100, 3)	1,728

Total params: 33,011,136 (125.93 MB)

Trainable params: 32,690,752 (124.71 MB)

Non-trainable params: 320,384 (1.22 MB)

Discriminator Summary:

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 50, 50, 64)	1,792
leaky_re_lu_10 (LeakyReLU)	(None, 50, 50, 64)	0
dropout_4 (Dropout)	(None, 50, 50, 64)	0
conv2d_5 (Conv2D)	(None, 25, 25, 128)	73,856
leaky_re_lu_11 (LeakyReLU)	(None, 25, 25, 128)	0
dropout_5 (Dropout)	(None, 25, 25, 128)	0
conv2d_6 (Conv2D)	(None, 13, 13, 256)	295,168
leaky_re_lu_12 (LeakyReLU)	(None, 13, 13, 256)	0
dropout_6 (Dropout)	(None, 13, 13, 256)	0
conv2d_7 (Conv2D)	(None, 7, 7, 512)	1,180,160
leaky_re_lu_13 (LeakyReLU)	(None, 7, 7, 512)	0
dropout_7 (Dropout)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_3 (Dense)	(None, 1)	25,089

```
Total params: 1,576,065 (6.01 MB)
Trainable params: 1,576,065 (6.01 MB)
Non-trainable params: 0 (0.00 B)
```

how to design and train a GAN from scratch, and what **noise, batch size, fixed noise**, and **training loop** mean.

---

1 Deciding the architecture

The GAN has **two networks**: Generator (G) and Discriminator (D).

Inputs you need to consider:

Parameter	Role	Guidelines
Image size	Determines initial shape in generator	Initial dense → reshape to `initial_size x initial_size x channels` where `initial_size = final_image_size / 2**n` (n = number of upsampling layers)
Image channels	Output channels in generator, input channels in discriminator	1 for grayscale, 3 for RGB
Batch size	Number of images processed together	Affects GPU memory and training stability; typical: 32–128
Latent dimension (noise size)	Size of input vector to generator	Typical: 50–200; higher = more variety but harder to train
Fixed noise	Noise vector used to monitor progress	Keep it constant so generated images can be compared across epochs

Example rules for 64×64 RGB images:

- Generator:
  - Dense → reshape `(8×8×256)`
  - Conv2DTranspose stride=2 → 16→32→64
  - Output: 64×64×3
- Discriminator:
  - Input: 64×64×3
  - Conv2D stride=2 → 32→16→8
  - Flatten → Dense(1)

2️⃣ Noise vs Fixed Noise

- **Noise**: Random latent vector `z ~ N(0,1)` or uniform `[-1,1]` passed to generator.
  - Every training step, you sample new noise for diversity.
- **Fixed Noise**: A **constant batch of latent vectors** used **only for monitoring**:
  - Same vectors every epoch
  - Allows you to see how generated images evolve
  - Example:

```
fixed_noise = tf.random.normal([25, LATENT_DIM])  # 25 images to monitor
```

3️⃣ Training functions

a. Losses

- Discriminator loss: real images → 1, fake images → 0

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss
```

- Generator loss: want fake images to fool discriminator

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

b. Optimizers

- Usually Adam with `learning_rate=1e-4` works well for both networks.
- Example:

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

4️⃣ Training step

```
@tf.function
def train_step(images):
    noise = tf.random.normal([batch_size, LATENT_DIM])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

    return gen_loss, disc_loss
```

5️⃣ Training loop

- Iterate over **epochs** and **batches**.
- Compute losses, update both networks.
- Optional: generate images from fixed noise to **monitor progress**.

```
for epoch in range(epochs):
    for image_batch in train_dataset:
        g_loss, d_loss = train_step(image_batch)
```

```
    # Display results every epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator, epoch, fixed_noise)
    plot_losses(gen_losses, disc_losses)
```

---

6 Summary of key design decisions

Decision	How to choose
Batch size	Depends on GPU; 32–128 typical
Noise size	50–200; bigger gives more variety
Fixed noise	Same every epoch, used for monitoring
Generator/Discriminator size	Rule of thumb: number of up/downsampling layers = log2(final_image_size / initial_size)
Loss	Binary cross entropy works for vanilla DCGAN
Optimizer	Adam(1e-4), beta1=0.5 commonly

---

key concept in GANs (and CNNs in general).

---

1 What is Up-sampling?

Up-sampling means **increasing the spatial dimensions (height × width) of an image or feature map**.

- In a **generator**, you start from a small feature map and "grow" it into a full image.
- Examples in GANs:
    1. **Conv2DTranspose** (sometimes called deconvolution)
        - Stride > 1 → increases size
    2. **UpSampling2D**
        - Simply repeats pixels to double the size

Example:

Layer	Input size	Output size
Dense → Reshape	(8×8×256)	8×8×256
Conv2DTranspose stride=2	8×8	16×16
Conv2DTranspose stride=2	16×16	32×32
Conv2DTranspose stride=2	32×32	64×64

So up-sampling = going from low-resolution → high-resolution

---

2 What is Down-sampling?

Down-sampling means **reducing the spatial dimensions (height × width)**.

- In a **discriminator**, you take a large image and reduce it to smaller feature maps to extract higher-level features.
- Examples in GANs:
    1. **Conv2D with stride > 1** (stride=2 halves size)
    2. **MaxPooling2D**

Example:

Layer	Input size	Output size
Conv2D stride=2	64×64	32×32
Conv2D stride=2	32×32	16×16
Conv2D stride=2	16×16	8×8

Layer	Input size	Output size
Flatten	8×8×256	16384 → Dense(1)

So down-sampling = going from high-resolution → low-resolution for easier classification

---

3️⃣ Why GANs need it

- **Generator:** needs up-sampling to transform small latent vectors into full images.
- **Discriminator:** needs down-sampling to reduce images to features and classify real vs fake.

Visual intuition

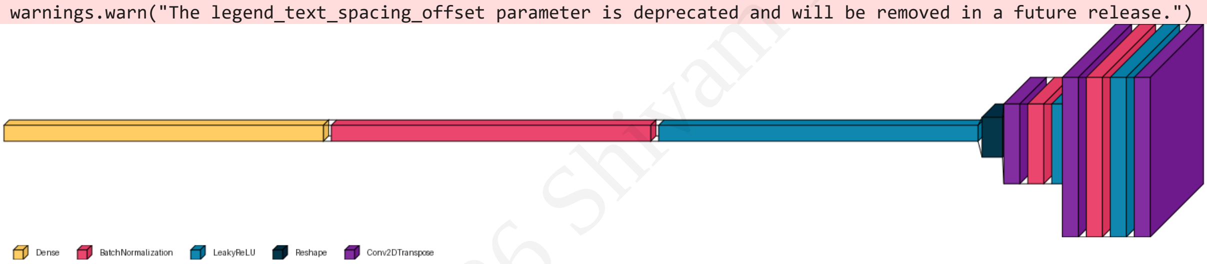
```
Generator: latent z → small feature map → ↑ ↑ ↑ → full image
Discriminator: image → ↓ ↓ ↓ → small vector → real/fake
```

- "↑" = up-sampling layers (Conv2DTranspose)
- "↓" = down-sampling layers (Conv2D stride>1)

In [8]:
```python
#visualize the models
import visualkeras
visualkeras.layered_view(generator, legend=True, to_file='generator.png', scale_xy=2)
```

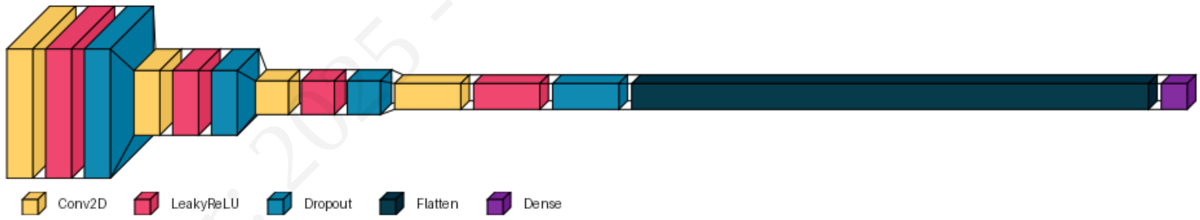
```
d:\venv\Lib\site-packages\visualkeras\layered.py:86: UserWarning: The legend_text_spacing_offset parameter is depre
cated and will be removed in a future release.
  warnings.warn("The legend_text_spacing_offset parameter is deprecated and will be removed in a future release.")
```

Out[8]:



In [9]:
```python
visualkeras.layered_view(discriminator, legend=True, to_file='discriminator.png', scale_xy=2)
```

Out[9]:



1️⃣ What is "Noise" in GANs?

- **Noise** is a **latent vector** (usually a 1D array) fed into the **generator**.
- It is **random**, sampled from a distribution, e.g., normal `N(0,1)` or uniform `[-1,1]`.
- Its purpose is to **introduce randomness and diversity** so the generator can produce different images each time.

Example:

```python
import tensorflow as tf

LATENT_DIM = 100
noise = tf.random.normal([32, LATENT_DIM])  # batch of 32 random vectors
```

- Shape `[batch_size, latent_dim]`
- `batch_size` = number of images per training step
- `latent_dim` = length of each noise vector (100 is common)

The generator learns to **map this noise vector into an image**.

-

2️⃣ Fixed Noise

- **Fixed noise** is just a **constant batch of latent vectors** that we use for **monitoring progress**.
- By feeding the **same vectors each epoch**, we can see how generated images evolve over time.

```
fixed_noise = tf.random.normal([25, LATENT_DIM])  # 25 images for monitoring
```

- This produces a **5×5 grid of images** every epoch using the same latent vectors.
- Without fixed noise, each epoch shows random images, making it hard to see training progress.

3️⃣ How to decide the latent vector size (LATENT_DIM)?

There's no strict rule, but some **guidelines**:

Latent dimension	Pros	Cons
Small (e.g., 16–50)	Easier to train, less memory	Low diversity, generator may produce similar images
Typical (e.g., 100)	Good diversity and stable training	Slightly higher computation
Large (e.g., 200+)	High diversity, can capture complex features	Harder to train, may cause instability

Rule of thumb:

- Start with **100** for MNIST or anime faces.
- Increase if your images are more complex (high-resolution, lots of features).

4️⃣ Recap

1. **Noise** = random latent vector → generator input → diversity in generated images
2. **Fixed noise** = same latent vector batch → track generator progress
3. **Size of noise (LATENT_DIM)** → usually 50–200 depending on dataset complexity

---

In [17]:
```
# ==========================================================
# 5. Training Setup
# ==========================================================
epochs = 300
noise_dim = LATENT_DIM
num_examples_to_generate = 25  # → 5x5 grid
fixed_noise = tf.random.normal([num_examples_to_generate, noise_dim])

os.makedirs("anime_generated", exist_ok=True)



data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
    layers.RandomContrast(0.1),
])

@tf.function # Decorator for performance helps to compile the function into a callable TensorFlow graph
def train_step(images):
    images = data_augmentation(images, training=True)
    noise = tf.random.normal([batch_size, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

    return gen_loss, disc_loss
```

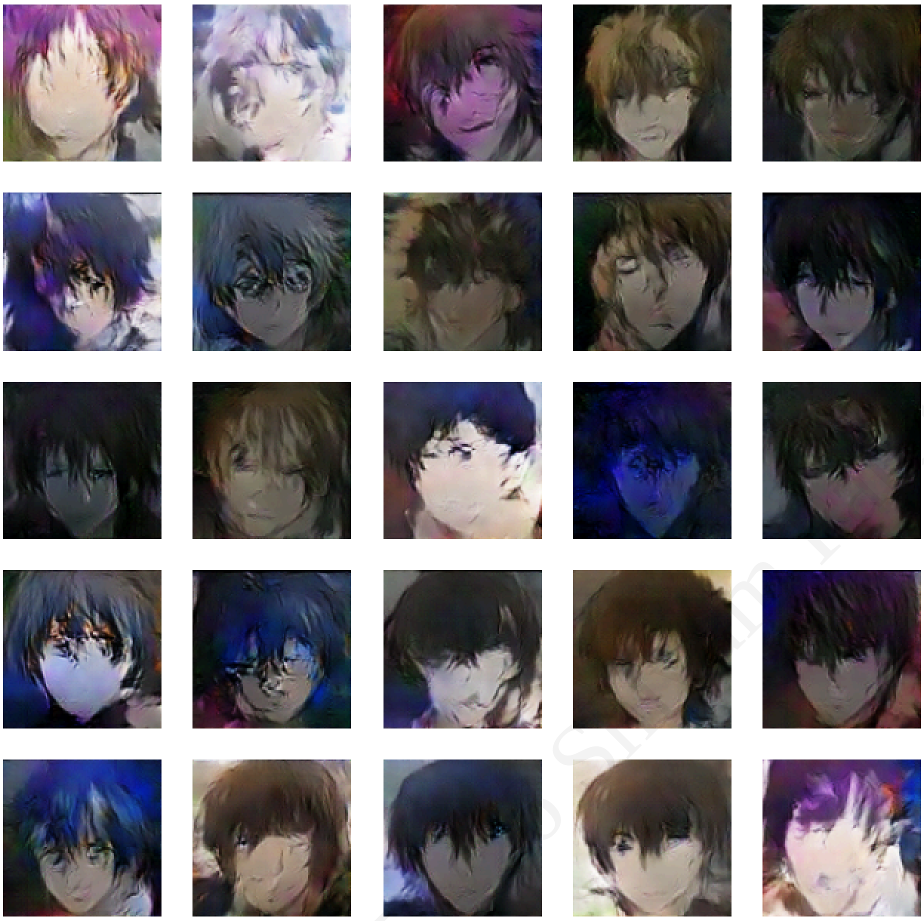
In [10]:
```
# ============================================================
# 6. Visualization Helpers
# ============================================================
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    predictions = (predictions + 1) / 2.0  # [-1,1] → [0,1]

    fig = plt.figure(figsize=(8, 8))
    for i in range(predictions.shape[0]):
        plt.subplot(5, 5, i+1)
        plt.imshow(predictions[i])
        plt.axis('off')
    plt.savefig(f"anime_generated/image_at_epoch_{epoch:04d}.png")
    plt.show()

def plot_losses(gen_losses, disc_losses):
    plt.figure(figsize=(8, 6))
    plt.plot(gen_losses, label=f'Generator Loss : {gen_losses[-1]:.4f}')
    plt.plot(disc_losses, label=f'Discriminator Loss : {disc_losses[-1]:.4f}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title("DCGAN Training Losses (Anime Faces)")
    plt.grid()
    plt.show()
```

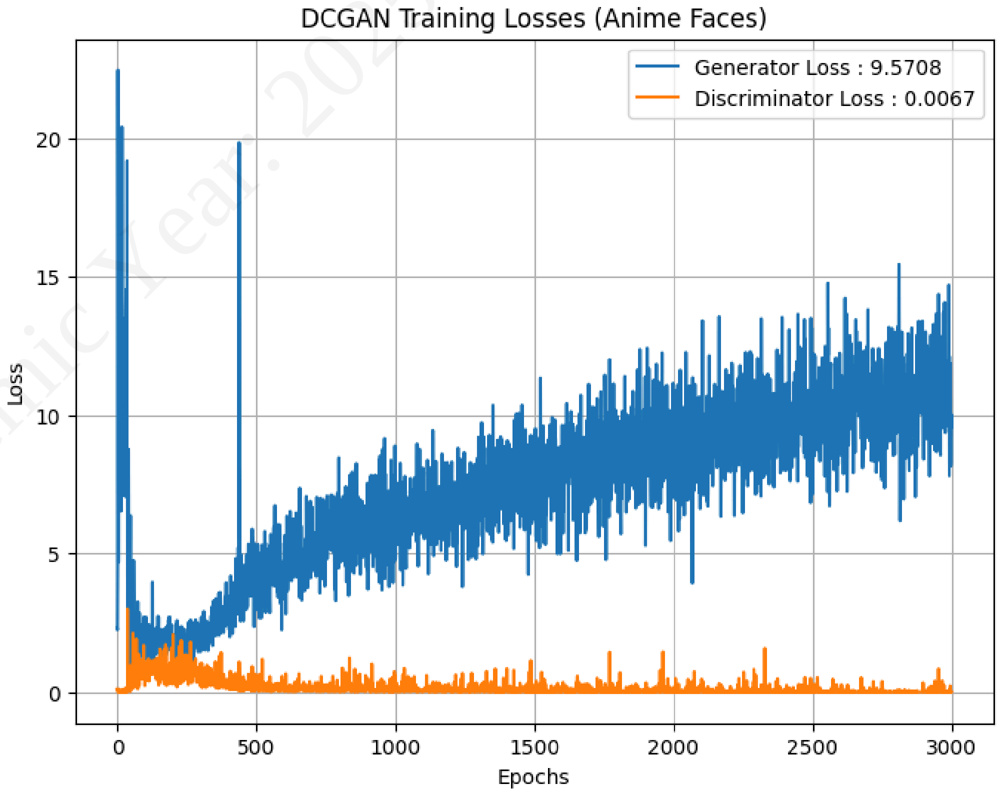
In [11]:
```
# ============================================================
# 7. Training Loop
# ============================================================
def train(dataset, epochs):
    for epoch in range(1, epochs+1):
        for image_batch in dataset:
            g_loss, d_loss = train_step(image_batch)

        gen_losses.append(g_loss.numpy())
        disc_losses.append(d_loss.numpy())

        # Inline display
        display.clear_output(wait=True)
        generate_and_save_images(generator, epoch, fixed_noise)
        plot_losses(gen_losses, disc_losses)

        print(f"Epoch {epoch}/{epochs} | Gen Loss: {g_loss:.4f}, Disc Loss: {d_loss:.4f}")

    # Final results
    generate_and_save_images(generator, epochs, fixed_noise)
    plot_losses(gen_losses, disc_losses)
```

In [12]:
```
# ============================================================
# 8. Create GIF from saved images
# ============================================================
def make_gif(image_folder="anime_generated", output_name="anime_dcgan.gif", duration=0.3):
    files = sorted(glob.glob(f"{image_folder}/*.png"))
    frames = [imageio.imread(f) for f in files]
    imageio.mimsave(output_name, frames, duration=duration)
    print(f"GIF saved as {output_name}")
    return output_name
```

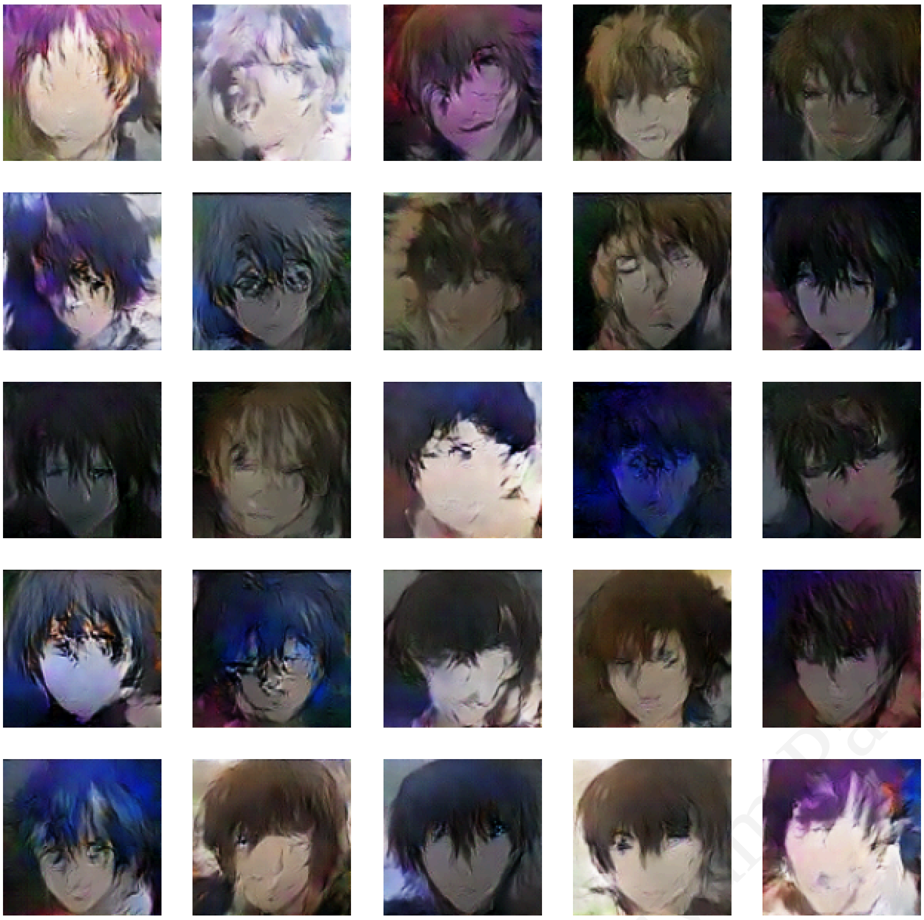
In [18]:
```
@tf.function # Decorator for performance helps to compile the function into a callable TensorFlow graph
def train_step(images):
    #images = data_augmentation(images, training=True) # no augmentation
    noise = tf.random.normal([batch_size, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

    return gen_loss, disc_loss
```

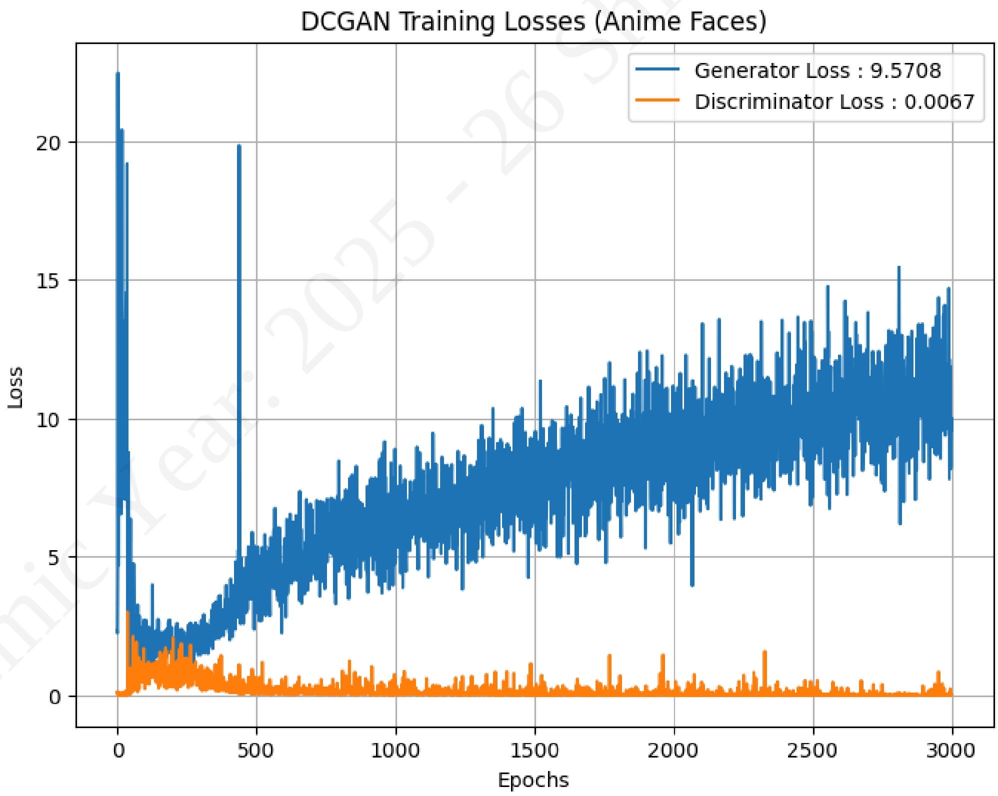
In [24]:
```python
# ============================================================
# 9. Run Training and Make GIF
# ============================================================
import imageio.v2 as imageio
epochs = 500
train(dataset, epochs)
anim_file = make_gif("anime_generated", "anime_dcgan.gif")
embed.embed_file(anim_file)
```





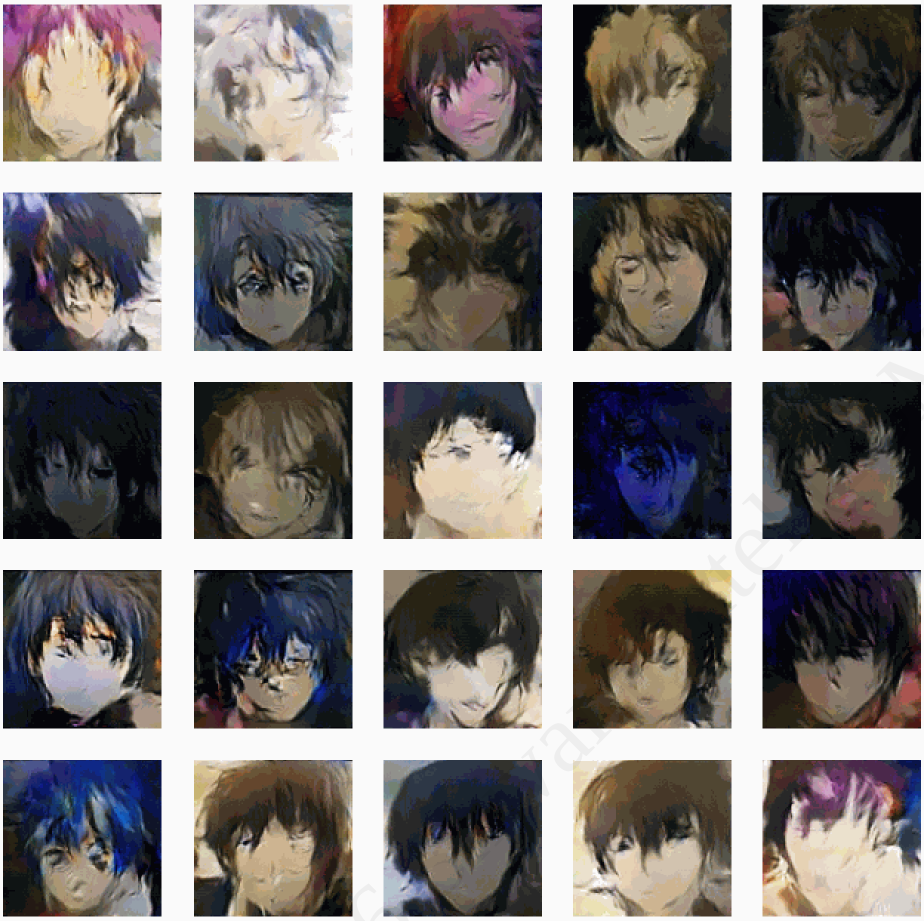

```
Epoch 500/500 | Gen Loss: 9.5708, Disc Loss: 0.0067
```






```
GIF saved as anime_dcgan.gif
```

Out[24]:



Various methods of saving Model

In [25]:
```
import os

# directory paths

checkpoint_dir = "training_checkpoints"

# Create directories if they don't exist

os.makedirs(checkpoint_dir, exist_ok=True)

ckpt = tf.train.Checkpoint(generator=generator,
                           discriminator=discriminator,
                           generator_optimizer=generator_optimizer,
                           discriminator_optimizer=discriminator_optimizer)

# Save
ckpt.save('training_checkpoints/ckpt')

# Later, restore
ckpt.restore(tf.train.latest_checkpoint('training_checkpoints')).expect_partial()
```

Out[25]: <tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x176ec5702f0>

In [26]:
```
import numpy as np

# After training, save the losses
np.save("gen_losses.npy", np.array(gen_losses))
np.save("disc_losses.npy", np.array(disc_losses))
```

```
# Later, to load them
gen_losses_loaded = np.load("gen_losses.npy")
disc_losses_loaded = np.load("disc_losses.npy")
```

In [29]:
```
import joblib

# Save
joblib.dump({'gen': gen_losses, 'disc': disc_losses}, 'gan_losses.joblib')

# Load
losses = joblib.load('gan_losses.joblib')
gen_losses_loaded = losses['gen']
disc_losses_loaded = losses['disc']
```

In [30]:
```
import tensorflow as tf

# Save model weights
gweights = generator.get_weights()
dweights = discriminator.get_weights()
joblib.dump({'gen_weights': gweights, 'disc_weights': dweights}, 'weights.joblib')

# Load back
weights_loaded = joblib.load('weights.joblib')
generator.set_weights(weights_loaded['gen_weights'])
discriminator.set_weights(weights_loaded['disc_weights'])
```

In [33]:
```
import keras


generator.save("generator_model.h5")
discriminator.save("discriminator_model.h5")
#discriminator.load_weights("discriminator_model.h5", compile=False)#
keras.saving.save_model(discriminator, 'discriminator.keras')
keras.saving.save_model(generator, 'generator.keras')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```