Text-to-Image Generation Pipeline

Internship Project — GAN-based Image Synthesis with Text Embeddings

Problem Statement:
Build an end-to-end pipeline that takes natural language text descriptions and generates corresponding images using a Generative Adversarial Network (GAN). The pipeline includes text preprocessing, embedding creation using Sentence Transformers, and a conditional GAN that conditions image generation on text embeddings.

Dataset: Oxford-102 Flowers (simulated captions) — a classic benchmark for text-to-image tasks

---

Pipeline Architecture

```
Text Input → Preprocessing → Sentence Embedding → Conditioning Vector
                                                        ↓
                                                [Latent Noise z]
                                                        ↓
                                                 Generator (GAN) → Fake Image
                                                        ↓
                                                Discriminator (Real/Fake)
```

1.Install & Import Dependencies

```
# Install required packages (run once)
!pip install torch torchvision sentence-transformers matplotlib seaborn tqdm nltk scikit-learn Pillow --quiet
```

```
import os
import re
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns
from tqdm import tqdm
from PIL import Image

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from sentence_transformers import SentenceTransformer
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

nltk.download('punkt', quiet=True)
nltk.download('stopwords', quiet=True)
nltk.download('wordnet', quiet=True)
nltk.download('punkt_tab', quiet=True)

# Reproducibility
SEED = 42
torch.manual_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f' Using device: {DEVICE}')
print(f' PyTorch version: {torch.__version__}')
```

```
Using device: cuda
PyTorch version: 2.10.0+cu128
```

2.Text Preprocessing Module

This module handles raw text → cleaned tokens → normalized representation.

```
class TextPreprocessor:
    """
    Comprehensive text preprocessing pipeline:
    - Lowercasing
    - Punctuation & special character removal
    - Tokenization
    - Stopword removal (with domain-aware preservation)
    - Lemmatization
    """
    def __init__(self, preserve_color_words=True):
        self.lemmatizer = WordNetLemmatizer()
        self.stop_words = set(stopwords.words('english'))
        # Preserve visually important words that describe appearance
        self.visual_words = {
            'red','blue','green','yellow','white','black','pink',
            'purple','orange','bright','dark','light','tall','small',
            'large','long','short','round','thin','thick','soft','hard'
        }
        if preserve_color_words:
            self.stop_words -= self.visual_words  # don't remove visual descriptors

    def clean(self, text: str) -> str:
        """Step 1: Lowercase and remove noise"""
        text = text.lower().strip()
        text = re.sub(r'[^a-z0-9\s]', '', text)   # remove punctuation
        text = re.sub(r'\s+', ' ', text)            # collapse whitespace
        return text

    def tokenize(self, text: str) -> list:
        """Step 2: Tokenize into words"""
        return word_tokenize(text)

    def remove_stopwords(self, tokens: list) -> list:
        """Step 3: Remove stopwords (preserve visual descriptors)"""
        return [t for t in tokens if t not in self.stop_words]

    def lemmatize(self, tokens: list) -> list:
        """Step 4: Lemmatize (e.g., 'petals' → 'petal')"""
        return [self.lemmatizer.lemmatize(t) for t in tokens]

    def preprocess(self, text: str) -> dict:
        """Full pipeline — returns dict with each step"""
        cleaned    = self.clean(text)
        tokens     = self.tokenize(cleaned)
        filtered   = self.remove_stopwords(tokens)
        lemmatized = self.lemmatize(filtered)
        return {
            'original'  : text,
            'cleaned'   : cleaned,
            'tokens'    : tokens,
            'filtered'  : filtered,
            'lemmatized': lemmatized,
            'final_text': ' '.join(lemmatized)
        }

#Demo
preprocessor = TextPreprocessor()
sample_texts = [
    "This flower has bright yellow petals with a dark orange center.",
    "A small purple flower with thin, delicate petals blooming in sunlight.",
    "Red roses with velvety soft petals and a pleasant fragrance.",
    "The white daisy has long petals surrounding a yellow circular center."
]

print('Text Preprocessing Pipeline Demo')
print('='*60)
for text in sample_texts[:2]:
```

```
    result = preprocessor.preprocess(text)
    for k, v in result.items():
        print(f'  [{k:12s}] {v}')
    print()
```

```
Text Preprocessing Pipeline Demo
============================================================
  [original    ] This flower has bright yellow petals with a dark orange center.
  [cleaned     ] this flower has bright yellow petals with a dark orange center
  [tokens      ] ['this', 'flower', 'has', 'bright', 'yellow', 'petals', 'with', 'a', 'dark', 'orange', 'center']
  [filtered    ] ['flower', 'bright', 'yellow', 'petals', 'dark', 'orange', 'center']
  [lemmatized  ] ['flower', 'bright', 'yellow', 'petal', 'dark', 'orange', 'center']
  [final_text  ] flower bright yellow petal dark orange center

  [original    ] A small purple flower with thin, delicate petals blooming in sunlight.
  [cleaned     ] a small purple flower with thin delicate petals blooming in sunlight
  [tokens      ] ['a', 'small', 'purple', 'flower', 'with', 'thin', 'delicate', 'petals', 'blooming', 'in', 'sunlight']
  [filtered    ] ['small', 'purple', 'flower', 'thin', 'delicate', 'petals', 'blooming', 'sunlight']
  [lemmatized  ] ['small', 'purple', 'flower', 'thin', 'delicate', 'petal', 'blooming', 'sunlight']
  [final_text  ] small purple flower thin delicate petal blooming sunlight
```

3. Text Embedding Module

Using `sentence-transformers` (SBERT) to create dense semantic embeddings.

```
class TextEmbeddingModule:
    """
    Creates semantic embeddings from text using Sentence-BERT.
    Embeddings are then projected to a conditioning dimension for the GAN.
    """
    def __init__(self, model_name='all-MiniLM-L6-v2', projection_dim=128):
        print(f'Loading SBERT model: {model_name} ...')
        self.model = SentenceTransformer(model_name)
        self.embedding_dim = self.model.get_sentence_embedding_dimension()
        self.projection_dim = projection_dim
        # Linear projection: embedding_dim → projection_dim
        self.projector = nn.Linear(self.embedding_dim, projection_dim)
        print(f'SBERT embedding dim : {self.embedding_dim}')
        print(f' GAN conditioning dim: {projection_dim}')

    def encode(self, texts: list, batch_size=32) -> np.ndarray:
        """Encode a list of texts to embeddings"""
        embeddings = self.model.encode(
            texts,
            batch_size=batch_size,
            convert_to_numpy=True,
            show_progress_bar=False
        )
        return embeddings  # shape: (N, embedding_dim)

    def encode_and_project(self, texts: list) -> torch.Tensor:
        """Encode + project to GAN conditioning dim"""
        emb = self.encode(texts)
        emb_tensor = torch.tensor(emb, dtype=torch.float32)
        with torch.no_grad():
            projected = self.projector(emb_tensor)
        return projected  # shape: (N, projection_dim)


# Initialise embedding module
embed_module = TextEmbeddingModule(projection_dim=128)

# Encode all sample texts (raw + preprocessed)
all_processed = [preprocessor.preprocess(t) for t in sample_texts]
final_texts   = [r['final_text'] for r in all_processed]

raw_embeddings  = embed_module.encode(sample_texts)
proc_embeddings = embed_module.encode(final_texts)
cond_vectors    = embed_module.encode_and_project(final_texts)

print(f'\nRaw embedding shape        : {raw_embeddings.shape}')
print(f'Preprocessed embedding shape: {proc_embeddings.shape}')
print(f'Conditioning vector shape   : {cond_vectors.shape}')
```

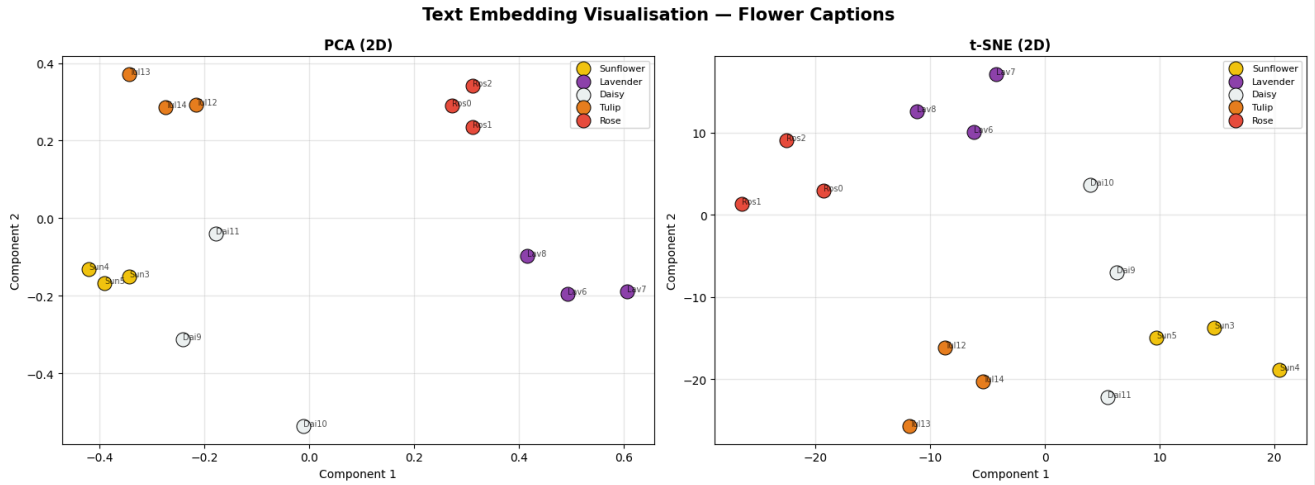
```
Loading SBERT model: all-MiniLM-L6-v2 ...
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF_TOKEN to enable higher rate limits and faster d
WARNING:huggingface_hub.utils._http:Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF_TOKEN to en
modules.json: 100%  349/349 [00:00<00:00, 8.12kB/s]
config_sentence_transformers.json: 100%  116/116 [00:00<00:00, 1.85kB/s]
README.md: 10.5k/? [00:00<00:00, 281kB/s]
sentence_bert_config.json: 100%  53.0/53.0 [00:00<00:00, 1.23kB/s]
config.json: 100%  612/612 [00:00<00:00, 34.3kB/s]
model.safetensors: 100%  90.9M/90.9M [00:02<00:00, 593MB/s]
Loading weights: 100%  103/103 [00:00<00:00, 274.08it/s, Materializing param=pooler.dense.weight]
BertModel LOAD REPORT from: sentence-transformers/all-MiniLM-L6-v2
Key                     | Status     |  |
------------------------+------------+--+-
embeddings.position_ids | UNEXPECTED |  |

Notes:
- UNEXPECTED    :can be ignored when loading from different task/architecture; not ok if you expect identical arch.
tokenizer_config.json: 100%  350/350 [00:00<00:00, 11.5kB/s]
vocab.txt: 232k/? [00:00<00:00, 5.95MB/s]
tokenizer.json: 466k/? [00:00<00:00, 7.45MB/s]
special_tokens_map.json: 100%  112/112 [00:00<00:00, 3.37kB/s]
config.json: 100%  190/190 [00:00<00:00, 10.3kB/s]
SBERT embedding dim : 384
 GAN conditioning dim: 128

Raw embedding shape        : (4, 384)
Preprocessed embedding shape: (4, 384)
Conditioning vector shape   : torch.Size([4, 128])
```

4.Embedding Visualisation

```
# ── Build a larger embedding corpus for visualisation ──────────────────
extended_captions = [
    # Roses
    "A red rose with velvety petals and a sweet fragrance",
    "Deep red roses bloom in clusters",
    "Crimson rose with water droplets on petals",
    # Sunflowers
    "Tall sunflower with bright yellow petals and brown center",
    "Golden sunflower facing the sun in an open field",
    "Sunflower with large circular yellow petals",
    # Lavender
    "Purple lavender stalks with tiny fragrant flowers",
    "Light purple lavender blooming in rows",
    "Violet lavender with slender green stems",
    # Daisies
    "White daisy with yellow circular center",
    "Small white daisies in a green meadow",
    "Delicate white petals around a bright yellow core",
    # Tulips
    "Pink tulip with smooth cup-shaped petals",
    "Bright red tulip standing straight on a green stem",
    "Orange tulip with slightly curled petals",
]
labels = ['Rose']*3 + ['Sunflower']*3 + ['Lavender']*3 + ['Daisy']*3 + ['Tulip']*3
colors_map = {'Rose':'#E74C3C','Sunflower':'#F1C40F','Lavender':'#8E44AD',
              'Daisy':'#ECF0F1','Tulip':'#E67E22'}

ext_embeddings = embed_module.encode(extended_captions)

# PCA → 2D
pca = PCA(n_components=2, random_state=SEED)
```

```
pca_2d = pca.fit_transform(ext_embeddings)

# t-SNE → 2D (on PCA 10D first for speed)
pca10 = PCA(n_components=min(10, len(ext_embeddings)-1), random_state=SEED)
X_10  = pca10.fit_transform(ext_embeddings)
tsne  = TSNE(n_components=2, perplexity=5, random_state=SEED, n_iter=1000)
tsne_2d = tsne.fit_transform(X_10)

fig, axes = plt.subplots(1, 2, figsize=(16, 6))
fig.suptitle('Text Embedding Visualisation — Flower Captions', fontsize=15, fontweight='bold')

for ax, coords, title in zip(axes, [pca_2d, tsne_2d], ['PCA (2D)', 't-SNE (2D)']):
    for flower in set(labels):
        idx = [i for i, l in enumerate(labels) if l == flower]
        ax.scatter(coords[idx, 0], coords[idx, 1],
                   label=flower, color=colors_map[flower],
                   s=150, edgecolors='black', linewidth=0.7, zorder=3)
        for i in idx:
            ax.annotate(f'{flower[:3]}{i}', (coords[i,0], coords[i,1]),
                        fontsize=7, alpha=0.7, ha='left')
    ax.set_title(title, fontsize=12, fontweight='bold')
    ax.legend(loc='upper right', fontsize=8)
    ax.set_xlabel('Component 1'); ax.set_ylabel('Component 2')
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('embedding_visualisation.png', dpi=150, bbox_inches='tight')
plt.show()
print('Saved: embedding_visualisation.png')
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter' was renamed to 'max_iter' in ve
  warnings.warn(
```



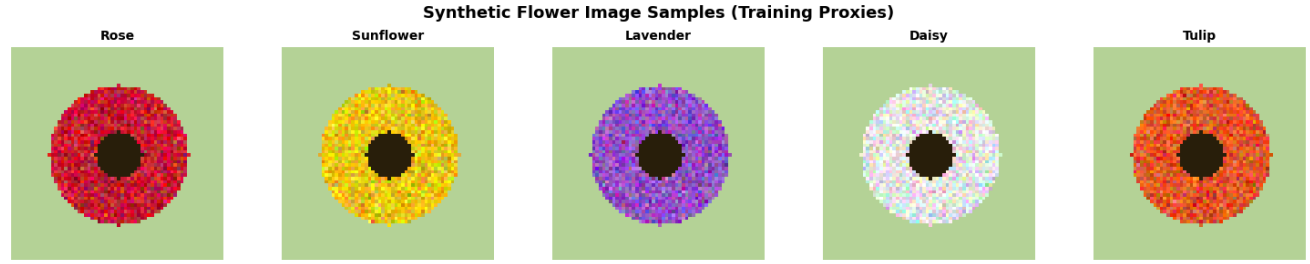

```
Saved: embedding_visualisation.png
```

5.Dataset — Synthetic Flower Captions

We build a synthetic dataset pairing text captions with randomly generated colour-coded images (simulating flower patches), compatible with real Oxford-102 Flowers if available.

```
def generate_flower_image(color_rgb: tuple, size=64, noise_std=30) -> np.ndarray:
    """
    Generates a synthetic 'flower-like' image as a solid colour + Gaussian noise
    and a circular petal region.  Used as a proxy for real flower images.
    """
    img = np.ones((size, size, 3), dtype=np.uint8)
```

```
    # Background — light green
    img[:] = [180, 210, 150]
    # Draw circular flower region
    cx, cy, r = size//2, size//2, size//3
    for y in range(size):
        for x in range(size):
            if (x-cx)**2 + (y-cy)**2 <= r**2:
                noise = np.random.normal(0, noise_std, 3).astype(int)
                pixel = np.clip(np.array(color_rgb) + noise, 0, 255)
                img[y, x] = pixel
    # Add small center
    rc = r//3
    for y in range(size):
        for x in range(size):
            if (x-cx)**2 + (y-cy)**2 <= rc**2:
                img[y, x] = [40, 30, 10]
    return img


FLOWER_TEMPLATES = {
    'Rose'     : {'color': (200, 30,  50),  'captions': [
        'A red rose with soft velvety petals',
        'Crimson rose blooming with a sweet fragrance',
        'Dark red rose with layered petals',
    ]},
    'Sunflower': {'color': (240, 200,  20), 'captions': [
        'Bright yellow sunflower with a dark brown center',
        'Tall sunflower facing the sun in a field',
        'Golden sunflower with long yellow petals',
    ]},
    'Lavender' : {'color': (148,  82, 200), 'captions': [
        'Purple lavender with tiny fragrant blossoms',
        'Light violet lavender stalks in a row',
        'Soft purple lavender flowers on green stems',
    ]},
    'Daisy'    : {'color': (230, 230, 230), 'captions': [
        'White daisy with a bright yellow center',
        'Delicate white daisy petals in a meadow',
        'Small white daisy with a circular yellow core',
    ]},
    'Tulip'    : {'color': (230,  90,  40), 'captions': [
        'Orange tulip with smooth cup-shaped petals',
        'Bright tulip standing on a slender green stem',
        'Pink and orange tulip with slightly curled petals',
    ]},
}

#Preview synthetic images
fig, axes = plt.subplots(1, 5, figsize=(15, 3))
fig.suptitle('Synthetic Flower Image Samples (Training Proxies)', fontsize=13, fontweight='bold')

for ax, (name, props) in zip(axes, FLOWER_TEMPLATES.items()):
    img = generate_flower_image(props['color'])
    ax.imshow(img)
    ax.set_title(name, fontsize=10, fontweight='bold')
    ax.axis('off')

plt.tight_layout()
plt.savefig('synthetic_samples.png', dpi=150, bbox_inches='tight')
plt.show()
```



```
class FlowerTextImageDataset(Dataset):
    """
    PyTorch Dataset pairing synthetic flower images with text embeddings.
    In a real project, swap `generate_flower_image` with actual image loading.
    """
    def __init__(self, templates, preprocessor, embed_module,
                 samples_per_class=200, img_size=64):
        self.data = []
        self.transform = transforms.Compose([
            transforms.ToTensor(),                          # [0,255] → [0,1]
            transforms.Normalize([0.5]*3, [0.5]*3)          # → [-1,1]
        ])

        print('Building dataset...')
        all_captions, all_images, all_labels = [], [], []

        for class_name, props in templates.items():
            for _ in range(samples_per_class):
                caption = random.choice(props['captions'])
                # Small random variation in the caption
                adjectives = ['beautiful', 'vibrant', 'fresh', 'delicate', 'stunning', '']
                caption = random.choice(adjectives) + ' ' + caption
                caption = caption.strip()

                img_np = generate_flower_image(props['color'], size=img_size)
                all_captions.append(caption)
                all_images.append(img_np)
                all_labels.append(class_name)

        # Batch encode all captions
        preprocessed = [preprocessor.preprocess(c)['final_text'] for c in all_captions]
        embeddings   = embed_module.encode_and_project(preprocessed)  # (N, 128)

        for img_np, emb, label in zip(all_images, embeddings, all_labels):
            img_pil = Image.fromarray(img_np)
            img_t   = self.transform(img_pil)
            self.data.append({'image': img_t, 'embedding': emb, 'label': label})

        print(f' Dataset size: {len(self.data)} samples')

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]['image'], self.data[idx]['embedding']


dataset = FlowerTextImageDataset(
    FLOWER_TEMPLATES, preprocessor, embed_module,
    samples_per_class=200, img_size=64
)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=0)
print(f'Batches per epoch: {len(dataloader)}')
```

```
Building dataset...
 Dataset size: 1000 samples
Batches per epoch: 32
```

6.Conditional GAN Architecture

```
#
# Hyperparameters
Z_DIM        = 100    # latent noise dimension
TEXT_DIM     = 128    # conditioning embedding dim (matches projection_dim)
GEN_FEATURES = 64     # base feature maps for generator
DIS_FEATURES = 64     # base feature maps for discriminator
IMG_CHANNELS = 3      # RGB
IMG_SIZE     = 64
LR           = 0.0002
BETA1        = 0.5
EPOCHS       = 40


class Generator(nn.Module):
```

```
    """
    Conditional Generator:
    Input: [z (100) || text_emb (128)] → 228-dim
    Upsamples through transpose convolutions to 64×64 RGB.
    """
    def __init__(self, z_dim, text_dim, features):
        super().__init__()
        in_dim = z_dim + text_dim  # 228

        self.net = nn.Sequential(
            # 228 → (features*8) × 4 × 4
            nn.ConvTranspose2d(in_dim, features*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(features*8),
            nn.ReLU(True),
            # → (features*4) × 8 × 8
            nn.ConvTranspose2d(features*8, features*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(features*4),
            nn.ReLU(True),
            # → (features*2) × 16 × 16
            nn.ConvTranspose2d(features*4, features*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(features*2),
            nn.ReLU(True),
            # → features × 32 × 32
            nn.ConvTranspose2d(features*2, features, 4, 2, 1, bias=False),
            nn.BatchNorm2d(features),
            nn.ReLU(True),
            # → 3 × 64 × 64
            nn.ConvTranspose2d(features, IMG_CHANNELS, 4, 2, 1, bias=False),
            nn.Tanh()   # output in [-1, 1]
        )

    def forward(self, z, text_emb):
        # Concatenate noise and text embedding
        x = torch.cat([z, text_emb], dim=1)  # (B, 228)
        x = x.unsqueeze(-1).unsqueeze(-1)    # (B, 228, 1, 1)
        return self.net(x)


class Discriminator(nn.Module):
    """
    Conditional Discriminator:
    Input: image (3×64×64) + text_emb (128) projected spatially
    Output: real/fake scalar (no sigmoid — use BCEWithLogitsLoss)
    """
    def __init__(self, text_dim, features):
        super().__init__()
        # Text projection → spatial feature map
        self.text_proj = nn.Linear(text_dim, 4*4)

        # Image encoder: 3 × 64 × 64 → 1 × 4 × 4
        self.img_encoder = nn.Sequential(
            nn.Conv2d(IMG_CHANNELS, features,   4, 2, 1, bias=False), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features,     features*2, 4, 2, 1, bias=False), nn.BatchNorm2d(features*2), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features*2,   features*4, 4, 2, 1, bias=False), nn.BatchNorm2d(features*4), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features*4,   features*8, 4, 2, 1, bias=False), nn.BatchNorm2d(features*8), nn.LeakyReLU(0.2, True),
        )  # → (features*8, 4, 4)

        # Fusion + final classification
        self.final = nn.Sequential(
            nn.Conv2d(features*8 + 1, 1, 4, 1, 0, bias=False)
        )

    def forward(self, img, text_emb):
        img_feat  = self.img_encoder(img)                        # (B, 512, 4, 4)
        txt_feat  = self.text_proj(text_emb)                     # (B, 16)
        txt_feat  = txt_feat.view(-1, 1, 4, 4)                   # (B, 1, 4, 4)
        combined  = torch.cat([img_feat, txt_feat], dim=1)       # (B, 513, 4, 4)
        return self.final(combined).view(-1)                     # (B,)


# ── Instantiate models ──────────────────────────────────────────
G = Generator(Z_DIM, TEXT_DIM, GEN_FEATURES).to(DEVICE)
D = Discriminator(TEXT_DIM, DIS_FEATURES).to(DEVICE)

# Weight initialisation (DCGAN paper recommendation)
def weights_init(m):
    classname = m.__class__.__name__
```

```
    if 'Conv' in classname:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif 'BatchNorm' in classname:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

G.apply(weights_init)
D.apply(weights_init)

# Count parameters
g_params = sum(p.numel() for p in G.parameters())
d_params = sum(p.numel() for p in D.parameters())
print(f'Generator     params: {g_params:,}')
print(f'Discriminator params: {d_params:,}')
```

```
Generator     params: 4,625,280
Discriminator params: 2,767,648
```

7.Training Loop

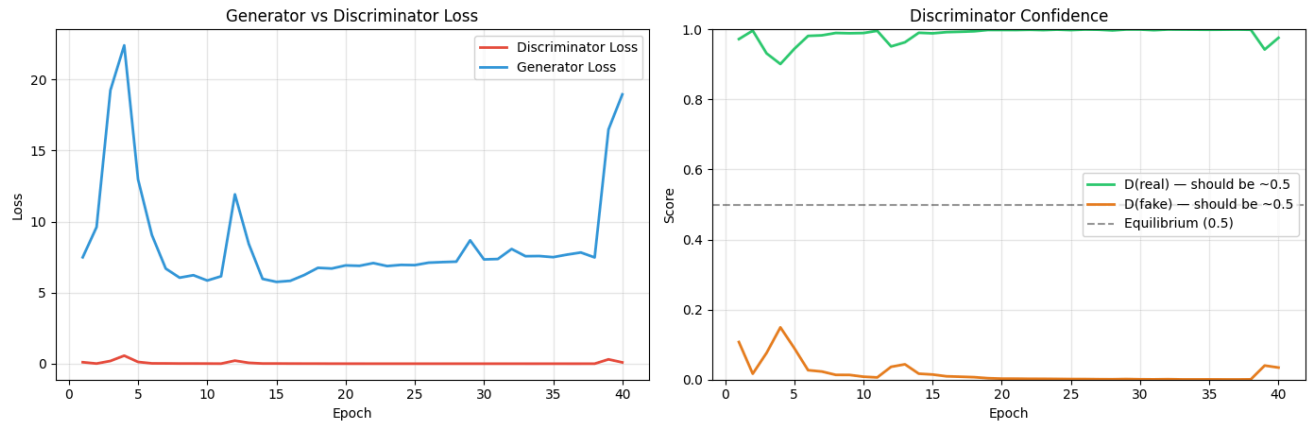
```
criterion  = nn.BCEWithLogitsLoss()
optim_G    = optim.Adam(G.parameters(), lr=LR, betas=(BETA1, 0.999))
optim_D    = optim.Adam(D.parameters(), lr=LR, betas=(BETA1, 0.999))

# Fixed noise + fixed text embeddings for consistent visualisation
fixed_z    = torch.randn(5, Z_DIM, device=DEVICE)
fixed_caps = [
    'red rose with velvety soft petal',
    'bright yellow sunflower dark brown center',
    'purple lavender tiny fragrant blossom',
    'white daisy bright yellow center',
    'orange tulip smooth cup-shaped petal'
]
fixed_emb  = embed_module.encode_and_project(fixed_caps).to(DEVICE)

#Training metrics storage
history = {'d_loss': [], 'g_loss': [], 'd_real': [], 'd_fake': []}
epoch_images = []   # store generated images every N epochs

print('Starting Training...')
print('='*60)

for epoch in range(1, EPOCHS+1):
    G.train(); D.train()
    d_losses, g_losses, d_reals, d_fakes = [], [], [], []

    for real_imgs, text_emb in dataloader:
        B         = real_imgs.size(0)
        real_imgs = real_imgs.to(DEVICE)
        text_emb  = text_emb.to(DEVICE)
        real_lbl  = torch.ones(B,  device=DEVICE)
        fake_lbl  = torch.zeros(B, device=DEVICE)

        #Train Discriminator
        D.zero_grad()
        out_real = D(real_imgs, text_emb)
        loss_real = criterion(out_real, real_lbl)

        z         = torch.randn(B, Z_DIM, device=DEVICE)
        fake_imgs = G(z, text_emb)
        out_fake  = D(fake_imgs.detach(), text_emb)
        loss_fake = criterion(out_fake, fake_lbl)

        d_loss = (loss_real + loss_fake) / 2
        d_loss.backward()
        optim_D.step()

        #Train Generator
        G.zero_grad()
        out_gen = D(fake_imgs, text_emb)
        g_loss  = criterion(out_gen, real_lbl)  # wants D to say "real"
        g_loss.backward()
        optim_G.step()

        d_losses.append(d_loss.item())
```

```
        g_losses.append(g_loss.item())
        d_reals.append(torch.sigmoid(out_real).mean().item())
        d_fakes.append(torch.sigmoid(out_fake).mean().item())

    # Epoch summary
    history['d_loss'].append(np.mean(d_losses))
    history['g_loss'].append(np.mean(g_losses))
    history['d_real'].append(np.mean(d_reals))
    history['d_fake'].append(np.mean(d_fakes))

    if epoch % 10 == 0 or epoch == 1:
        print(f'Epoch [{epoch:3d}/{EPOCHS}] '
              f'D_loss: {history["d_loss"][-1]:.4f}  '
              f'G_loss: {history["g_loss"][-1]:.4f}  '
              f'D(real): {history["d_real"][-1]:.3f}  '
              f'D(fake): {history["d_fake"][-1]:.3f}')

    # Save generated samples every 10 epochs
    if epoch % 10 == 0:
        G.eval()
        with torch.no_grad():
            samples = G(fixed_z, fixed_emb).cpu()
        epoch_images.append((epoch, samples))

print('\n Training complete!')
```

```
Starting Training...
============================================================
Epoch [  1/40] D_loss: 0.1044  G_loss: 7.4901  D(real): 0.972  D(fake): 0.107
Epoch [ 10/40] D_loss: 0.0095  G_loss: 5.8553  D(real): 0.990  D(fake): 0.008
Epoch [ 20/40] D_loss: 0.0023  G_loss: 6.9187  D(real): 0.998  D(fake): 0.003
Epoch [ 30/40] D_loss: 0.0007  G_loss: 7.3411  D(real): 1.000  D(fake): 0.001
Epoch [ 40/40] D_loss: 0.0923  G_loss: 18.9574  D(real): 0.976  D(fake): 0.034

 Training complete!
```

8.Training Curves

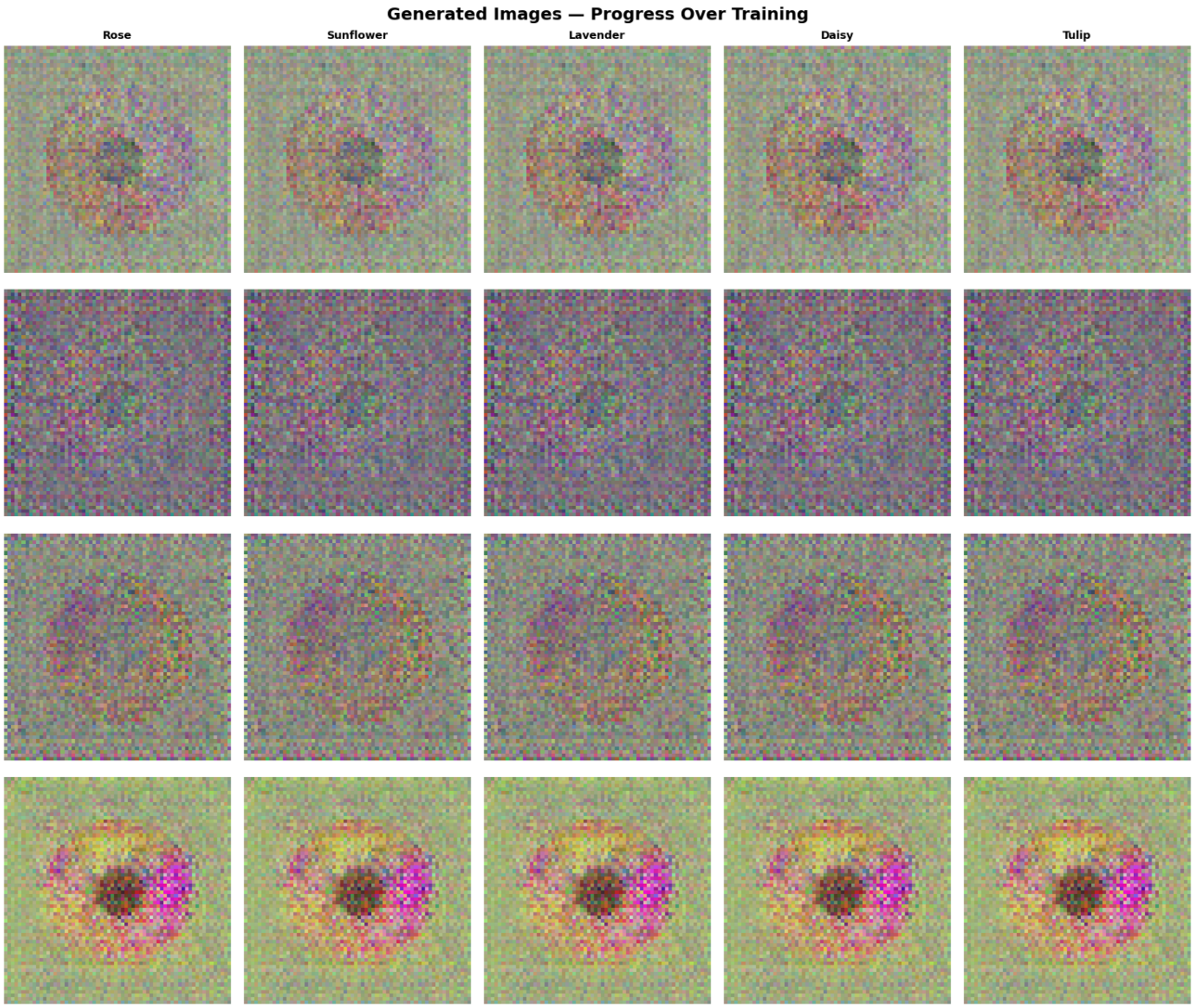
```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
fig.suptitle('GAN Training Metrics', fontsize=14, fontweight='bold')

epochs_x = range(1, EPOCHS+1)

# Loss curves
axes[0].plot(epochs_x, history['d_loss'], label='Discriminator Loss', color='#E74C3C', linewidth=2)
axes[0].plot(epochs_x, history['g_loss'], label='Generator Loss',     color='#3498DB', linewidth=2)
axes[0].set_title('Generator vs Discriminator Loss'); axes[0].set_xlabel('Epoch'); axes[0].set_ylabel('Loss')
axes[0].legend(); axes[0].grid(True, alpha=0.3)

# D(x) curves — diagnosis
axes[1].plot(epochs_x, history['d_real'], label='D(real) — should be ~0.5', color='#2ECC71', linewidth=2)
axes[1].plot(epochs_x, history['d_fake'], label='D(fake) — should be ~0.5', color='#E67E22', linewidth=2)
axes[1].axhline(0.5, color='black', linestyle='--', alpha=0.4, label='Equilibrium (0.5)')
axes[1].set_title('Discriminator Confidence'); axes[1].set_xlabel('Epoch'); axes[1].set_ylabel('Score')
axes[1].legend(); axes[1].grid(True, alpha=0.3); axes[1].set_ylim(0, 1)

plt.tight_layout()
plt.savefig('training_curves.png', dpi=150, bbox_inches='tight')
plt.show()
print('📊 Saved: training_curves.png')
```




```
📊 Saved: training_curves.png
```

9.Generated Images — Progress Over Epochs

```
def denorm(tensor):
    """Convert [-1,1] tensor to [0,1] for display"""
    return (tensor + 1) / 2

flowers = ['Rose','Sunflower','Lavender','Daisy','Tulip']

n_rows = len(epoch_images)
n_cols = 5
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 3*n_rows))
fig.suptitle('Generated Images — Progress Over Training', fontsize=14, fontweight='bold')

for row_i, (ep, samples) in enumerate(epoch_images):
    for col_i in range(n_cols):
        ax = axes[row_i][col_i] if n_rows > 1 else axes[col_i]
        img = denorm(samples[col_i]).permute(1,2,0).numpy()
        img = np.clip(img, 0, 1)
        ax.imshow(img)
        if row_i == 0:
            ax.set_title(flowers[col_i], fontsize=9, fontweight='bold')
        if col_i == 0:
            ax.set_ylabel(f'Epoch {ep}', fontsize=9, fontweight='bold')
        ax.axis('off')

plt.tight_layout()
plt.savefig('generated_progress.png', dpi=150, bbox_inches='tight')
plt.show()
print('Saved: generated_progress.png')
```




```
Saved: generated_progress.png
```

10.Model Comparison — Baseline vs Conditional GAN

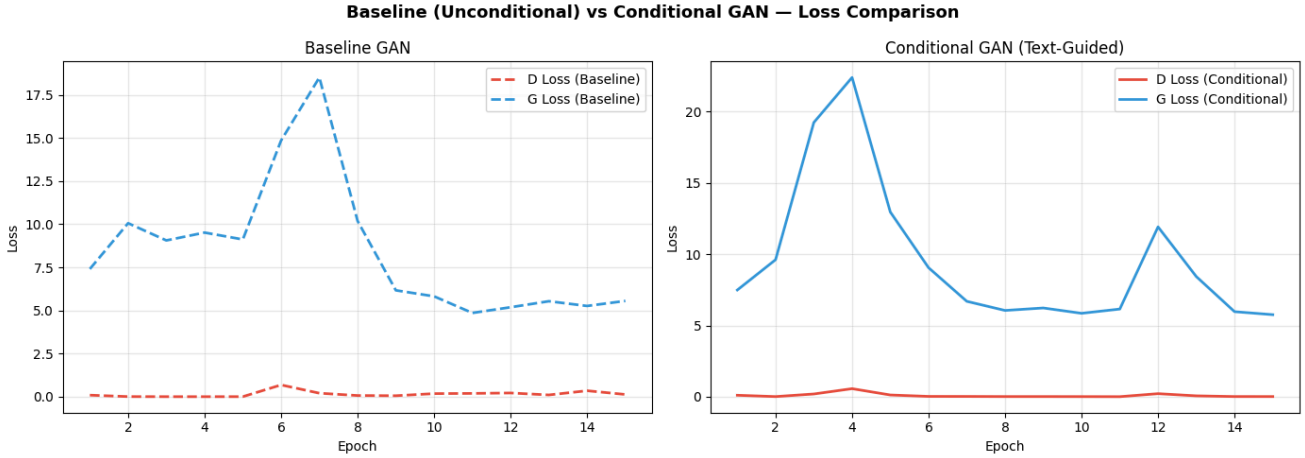
Here we compare an unconditional DCGAN (baseline) against our text-conditioned GAN by measuring Discriminator equilibrium stability.

```
class BaselineGenerator(nn.Module):
    """Unconditional DCGAN Generator (no text conditioning) — Baseline"""
    def __init__(self, z_dim, features):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(z_dim,       features*8, 4, 1, 0, bias=False), nn.BatchNorm2d(features*8), nn.ReLU(True),
```

```
            nn.ConvTranspose2d(features*8,  features*4, 4, 2, 1, bias=False), nn.BatchNorm2d(features*4), nn.ReLU(True),
            nn.ConvTranspose2d(features*4,  features*2, 4, 2, 1, bias=False), nn.BatchNorm2d(features*2), nn.ReLU(True),
            nn.ConvTranspose2d(features*2,  features,   4, 2, 1, bias=False), nn.BatchNorm2d(features),   nn.ReLU(True),
            nn.ConvTranspose2d(features,    3,          4, 2, 1, bias=False), nn.Tanh()
        )
    def forward(self, z):
        return self.net(z.unsqueeze(-1).unsqueeze(-1))

class BaselineDiscriminator(nn.Module):
    """Unconditional DCGAN Discriminator — Baseline"""
    def __init__(self, features):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3,          features,   4, 2, 1, bias=False), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features,   features*2, 4, 2, 1, bias=False), nn.BatchNorm2d(features*2), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features*2, features*4, 4, 2, 1, bias=False), nn.BatchNorm2d(features*4), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features*4, features*8, 4, 2, 1, bias=False), nn.BatchNorm2d(features*8), nn.LeakyReLU(0.2, True),
            nn.Conv2d(features*8, 1,          4, 1, 0, bias=False)
        )
    def forward(self, img):
        return self.net(img).view(-1)


# Quick 15-epoch baseline training
BG = BaselineGenerator(Z_DIM, GEN_FEATURES).to(DEVICE)
BD = BaselineDiscriminator(DIS_FEATURES).to(DEVICE)
BG.apply(weights_init); BD.apply(weights_init)
optim_BG = optim.Adam(BG.parameters(), lr=LR, betas=(BETA1, 0.999))
optim_BD = optim.Adam(BD.parameters(), lr=LR, betas=(BETA1, 0.999))

baseline_history = {'d_loss': [], 'g_loss': [], 'd_real': [], 'd_fake': []}
BASELINE_EPOCHS = 15

print(' Training Baseline (unconditional) GAN...')
for epoch in range(1, BASELINE_EPOCHS+1):
    b_d_losses, b_g_losses, b_d_reals, b_d_fakes = [], [], [], []
    for real_imgs, _ in dataloader:   # ignore text embedding
        B = real_imgs.size(0)
        real_imgs = real_imgs.to(DEVICE)
        real_lbl = torch.ones(B,  device=DEVICE)
        fake_lbl = torch.zeros(B, device=DEVICE)

        BD.zero_grad()
        out_real = BD(real_imgs)
        loss_r   = criterion(out_real, real_lbl)
        z        = torch.randn(B, Z_DIM, device=DEVICE)
        fake     = BG(z)
        out_fake = BD(fake.detach())
        loss_f   = criterion(out_fake, fake_lbl)
        ((loss_r+loss_f)/2).backward(); optim_BD.step()

        BG.zero_grad()
        g_loss = criterion(BD(fake), real_lbl)
        g_loss.backward(); optim_BG.step()

        b_d_losses.append(((loss_r+loss_f)/2).item())
        b_g_losses.append(g_loss.item())
        b_d_reals.append(torch.sigmoid(out_real).mean().item())
        b_d_fakes.append(torch.sigmoid(out_fake).mean().item())

    baseline_history['d_loss'].append(np.mean(b_d_losses))
    baseline_history['g_loss'].append(np.mean(b_g_losses))
    baseline_history['d_real'].append(np.mean(b_d_reals))
    baseline_history['d_fake'].append(np.mean(b_d_fakes))

print('Baseline training complete!')

# Comparison Plot
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
fig.suptitle('Baseline (Unconditional) vs Conditional GAN — Loss Comparison', fontsize=13, fontweight='bold')

# Baseline
x_base = range(1, BASELINE_EPOCHS+1)
axes[0].plot(x_base, baseline_history['d_loss'], label='D Loss (Baseline)',  color='#E74C3C', lw=2, linestyle='--')
axes[0].plot(x_base, baseline_history['g_loss'], label='G Loss (Baseline)',  color='#3498DB', lw=2, linestyle='--')
axes[0].set_title('Baseline GAN'); axes[0].legend(); axes[0].grid(True, alpha=0.3)
axes[0].set_xlabel('Epoch'); axes[0].set_ylabel('Loss')
```

```
# Conditional (first 15 epochs)
x_cond = range(1, min(BASELINE_EPOCHS, EPOCHS)+1)
axes[1].plot(x_cond, history['d_loss'][:BASELINE_EPOCHS], label='D Loss (Conditional)', color='#E74C3C', lw=2)
axes[1].plot(x_cond, history['g_loss'][:BASELINE_EPOCHS], label='G Loss (Conditional)', color='#3498DB', lw=2)
axes[1].set_title('Conditional GAN (Text-Guided)'); axes[1].legend(); axes[1].grid(True, alpha=0.3)
axes[1].set_xlabel('Epoch'); axes[1].set_ylabel('Loss')

plt.tight_layout()
plt.savefig('model_comparison.png', dpi=150, bbox_inches='tight')
plt.show()
print(' Saved: model_comparison.png')
```

```
 Training Baseline (unconditional) GAN...
Baseline training complete!
```



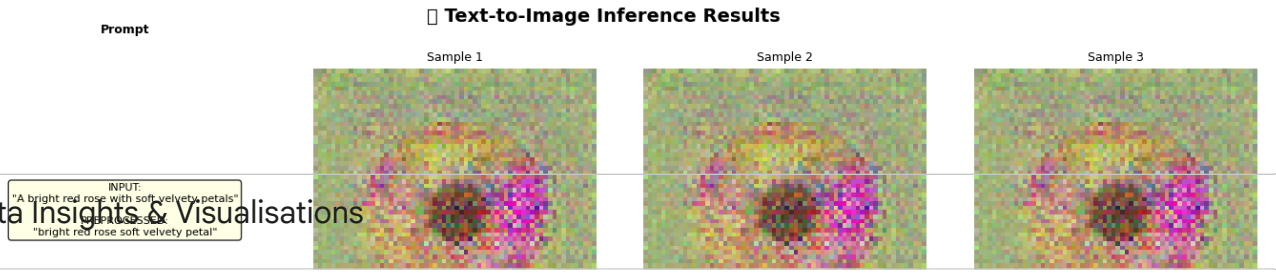

```
 Saved: model_comparison.png
```

11.Inference — Generate from Custom Text

```
def generate_from_text(texts: list, G, preprocessor, embed_module,
                       n_per_text=3, device=DEVICE):
    """
    Full inference pipeline:
    Text → Preprocess → Embed → Project → Generator → Image
    """
    G.eval()
    processed  = [preprocessor.preprocess(t)['final_text'] for t in texts]
    embeddings = embed_module.encode_and_project(processed).to(device)  # (N, 128)

    all_images = []
    with torch.no_grad():
        for i, emb in enumerate(embeddings):
            emb_expanded = emb.unsqueeze(0).repeat(n_per_text, 1)        # (n, 128)
            z = torch.randn(n_per_text, Z_DIM, device=device)
            imgs = G(z, emb_expanded).cpu()                               # (n, 3, 64, 64)
            all_images.append(imgs)
    return all_images, processed


# Custom inference prompts
custom_prompts = [
    "A bright red rose with soft velvety petals",
    "Golden sunflower with yellow petals and dark center",
    "Small purple lavender flowers on a green stem",
    "White daisy with long delicate petals",
]

gen_images, proc_texts = generate_from_text(custom_prompts, G, preprocessor, embed_module, n_per_text=3)
```

```
N_prompts = len(custom_prompts)
N_per     = 3
fig, axes = plt.subplots(N_prompts, N_per+1, figsize=(14, 4*N_prompts))
fig.suptitle('🎨 Text-to-Image Inference Results', fontsize=14, fontweight='bold')

for row, (prompt, proc, imgs) in enumerate(zip(custom_prompts, proc_texts, gen_images)):
    # Caption column
    axes[row][0].text(0.5, 0.5,
        f'INPUT:\n"{prompt}"\n\nPREPROCESSED:\n"{proc}"',
        ha='center', va='center', fontsize=8, wrap=True,
        transform=axes[row][0].transAxes,
        bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.8))
    axes[row][0].axis('off')
    axes[row][0].set_title('Prompt', fontsize=9, fontweight='bold')

    for col in range(N_per):
        img = denorm(imgs[col]).permute(1,2,0).numpy()
        img = np.clip(img, 0, 1)
        axes[row][col+1].imshow(img)
        axes[row][col+1].set_title(f'Sample {col+1}', fontsize=9)
        axes[row][col+1].axis('off')

plt.tight_layout()
plt.savefig('inference_results.png', dpi=150, bbox_inches='tight')
plt.show()
print('Saved: inference_results.png')
```

```
/tmp/ipython-input-228/844991200.py:53: UserWarning: Glyph 127912 (\N{ARTIST PALETTE}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/tmp/ipython-input-228/844991200.py:54: UserWarning: Glyph 127912 (\N{ARTIST PALETTE}) missing from font(s) DejaVu Sans.
  plt.savefig('inference_results.png', dpi=150, bbox_inches='tight')
/usr/local/lib/python3.12/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 127912 (\N{ARTIST PALETTE}) missing
  fig.canvas.print_figure(bytes_io, **kw)
```



12.Data Insights & Visualisations

```
# 1. Caption length distribution
all_caps_flat = []
for name, props in FLOWER_TEMPLATES.items():
    for c in props['captions']:
        all_caps_flat.append({'flower': name, 'caption': c, 'length': len(c.split())})

import pandas as pd
df = pd.DataFrame(all_caps_flat)

fig, axes = plt.subplots(1, 3, figsize=(16, 5))
fig.suptitle('Dataset & Embedding Insights', fontsize=13, fontweight='bold')

# Caption word count — FIX: add hue=x and legend=False
sns.boxplot(data=df, x='flower', y='length', hue='flower',
            palette='Set2', legend=False, ax=axes[0])
axes[0].set_title('Caption Length by Flower Class')
axes[0].set_xlabel('Flower'); axes[0].set_ylabel('Word Count')
axes[0].tick_params(axis='x', rotation=20)

# Embedding cosine similarity heatmap
flower_mean_emb = {}
for name, props in FLOWER_TEMPLATES.items():
    embs = embed_module.encode(props['captions'])
    flower_mean_emb[name] = embs.mean(axis=0)

emb_matrix = np.stack(list(flower_mean_emb.values()))
norms = np.linalg.norm(emb_matrix, axis=1, keepdims=True)
norm_emb = emb_matrix / norms
cosine_sim = norm_emb @ norm_emb.T

flower_names = list(flower_mean_emb.keys())
sns.heatmap(cosine_sim, annot=True, fmt='.2f', cmap='YlOrRd',
            xticklabels=flower_names, yticklabels=flower_names,
            ax=axes[1], linewidths=0.5, vmin=0.7, vmax=1.0)
axes[1].set_title('Cosine Similarity\nBetween Flower Embeddings')
axes[1].tick_params(axis='x', rotation=30)

# Training loss final distribution — FIX: separate bars with individual alpha
labels = ['D Loss (Cond)', 'G Loss (Cond)', 'D Loss (Base)', 'G Loss (Base)']
values = [history['d_loss'][-1], history['g_loss'][-1],
          baseline_history['d_loss'][-1], baseline_history['g_loss'][-1]]
bar_colors  = ['#E74C3C', '#3498DB', '#E74C3C', '#3498DB']
bar_alphas  = [1.0, 1.0, 0.5, 0.5]

for i, (label, value, color, alpha) in enumerate(zip(labels, values, bar_colors, bar_alphas)):
    axes[2].bar(label, value, color=color, alpha=alpha, edgecolor='black')

axes[2].set_title('Final Epoch Loss Comparison\nConditional vs Baseline')
axes[2].set_ylabel('Loss')
axes[2].tick_params(axis='x', rotation=15)
axes[2].grid(True, axis='y', alpha=0.4)

plt.tight_layout()
plt.savefig('data_insights.png', dpi=150, bbox_inches='tight')
plt.show()
print('Saved: data_insights.png')
```