Project Title: "Linguist-to-Lens"

A Text-to-Image Generation Pipeline using CGANs

1. Problem Statement

The goal is to build a system that can take a text description (e.g., "a red flower") and generate a corresponding 64×64 pixel image.

The Challenge: Computers don't "see" text or images the same way. We must bridge the gap by:

1. Converting text into numerical vectors (Embeddings).
2. Training a Generator to turn those vectors into pixels.
3. Training a Discriminator to ensure those pixels look realistic and match the text.

2. Dataset & Preprocessing

For a beginner-to-intermediate project, we recommend using the Oxford-102 Flowers or CUB-200 Birds datasets. They contain images paired with high-quality text descriptions.

A. Data Handling

- Image Processing: Resize images to a uniform size (e.g., 64x64), normalize pixel values to $[-1, 1]$, and handle any corrupted files.
- Text Preprocessing: Use NLTK or spaCy to clean descriptions (lowercase, removing punctuation).

B. Text Embedding Creation

To make the text "understandable" for the GAN, we use a pre-trained model to create Embeddings.

- Tool: Sentence-Transformers (BERT-based).
- Logic: Every sentence is converted into a fixed-length vector (e.g., 768 dimensions).

3. Modeling: The GAN Architecture

We will implement a Conditional GAN (CGAN). Unlike a standard GAN, a CGAN takes a "condition" (our text embedding) to guide the generation process.

The Pipeline Components:

1. The Embedding Layer: Transforms the text into a dense vector.
2. The Generator: A "Deconvolutional" neural network. It takes the text vector + random noise and "upsamples" them into an image.
3. The Discriminator: A "Convolutional" neural network. It looks at an image and the text vector together and decides: *"Is this a real image that matches the text, or a fake?"*

$$Loss = \mathbb{E}_{x \sim p_{data}}[\log D(x|y)] + \mathbb{E}_{z \sim p_{z}}[\log(1 - D(G(z|y)|y))]$$

4. Technical Implementation (The Notebook)

Phase	Tasks	Tools
EDA	Plot distribution of text lengths; visualize sample images.	Matplotlib, Seaborn
Preprocessing	Normalize images; Tokenize text; Create a PyTorch/TF DataLoader.	NumPy, Pandas
Training	Run the GAN loops (Train D, then Train G). Monitor "Loss" curves.	Scikit-learn, PyTorch/Keras
Evaluation	Use Inception Score (IS) or visual inspection of generated grids.	Matplotlib

5. Results & Insights

- Model Performance: You will observe the "Minimax" game where the Generator and Discriminator improve together.
- Visual Evolution: Document how the images move from "colored noise" at Epoch 1 to "recognizable shapes" at Epoch 100.
- Limitations: Discuss why GANs sometimes suffer from "Mode Collapse" (generating the same image repeatedly).

6. Documentation & Submission

- README: Explain how to install dependencies (pip install -r requirements.txt).
- Comments: Use Markdown cells in your Jupyter Notebook to explain *why* you chose specific hyperparameters (like learning rate).

- **Git: Commit your code frequently with clear messages (e.g., "Added text embedding logic").**

Python

```
import torch
from sentence_transformers import SentenceTransformer


# 1. Initialize the embedding model (e.g., MiniLM - fast and lightweight)
text_encoder = SentenceTransformer('all-MiniLM-L6-v2')


def create_embeddings(descriptions):
    """
    Converts a list of text strings into a tensor of numerical embeddings.
    """
    # Generate embeddings (shape: [num_sentences, 384])
    embeddings = text_encoder.encode(descriptions)
    return torch.tensor(embeddings).float()


# Example Usage:
descriptions = ["a beautiful red rose", "a yellow sunflower in a field"]
text_vectors = create_embeddings(descriptions)
print(f"Embedding Shape: {text_vectors.shape}") # [2, 384]
```

Phase 2: The Generator & Discriminator (Architecture)

The Generator takes the text embedding + random noise to create an image. The Discriminator evaluates if the image matches the text description.

Python

```
import torch.nn as nn


class Generator(nn.Module):
```

```
def __init__(self, embedding_dim, latent_dim=100):
    super(Generator, self).__init__()
    # Combine noise and text embedding
    self.init_size = 64 // 4
    self.l1 = nn.Sequential(nn.Linear(latent_dim + embedding_dim, 128 * self.init_size**2))

    self.conv_blocks = nn.Sequential(
        nn.BatchNorm2d(128),
        nn.Upsample(scale_factor=2),
        nn.Conv2d(128, 128, 3, stride=1, padding=1),
        nn.BatchNorm2d(128, 0.8),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Upsample(scale_factor=2),
        nn.Conv2d(128, 64, 3, stride=1, padding=1),
        nn.BatchNorm2d(64, 0.8),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(64, 3, 3, stride=1, padding=1), # 3 channels (RGB)
        nn.Tanh(),
    )

def forward(self, noise, text_embeddings):
    # Concatenate noise and text info
    gen_input = torch.cat((noise, text_embeddings), -1)
    out = self.l1(gen_input)
    out = out.view(out.shape[0], 128, self.init_size, self.init_size)
    img = self.conv_blocks(out)
```

```
        return img


class Discriminator(nn.Module):
    def __init__(self, embedding_dim):
        super(Discriminator, self).__init__()
        # Reduced CNN for simplicity
        self.model = nn.Sequential(
            nn.Conv2d(3, 16, 3, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.25),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(32, 0.8),
        )
        # Final layer also takes text embedding into account
        self.adv_layer = nn.Sequential(nn.Linear(32 * 16 * 16 + embedding_dim, 1), nn.Sigmoid())


    def forward(self, img, text_embeddings):
        out = self.model(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(torch.cat((out, text_embeddings), -1))
        return validity
```

Phase 3: The Training Loop (Logic)

This follows the "Correctness of Logic" evaluation criteria.

Python

```
# Hyperparameters
```

```
latent_dim = 100

embedding_dim = 384 # Matches MiniLM output

lr = 0.0002


# Initialize models and optimizers

generator = Generator(embedding_dim, latent_dim)

discriminator = Discriminator(embedding_dim)

optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr)

optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr)

adversarial_loss = nn.BCELoss()


# Training step simulation

def train_step(real_imgs, text_embeddings):

    batch_size = real_imgs.size(0)


    # 1. Train Generator

    optimizer_G.zero_grad()

    z = torch.randn(batch_size, latent_dim) # Random noise

    gen_imgs = generator(z, text_embeddings)

    g_loss = adversarial_loss(discriminator(gen_imgs, text_embeddings),
torch.ones(batch_size, 1))

    g_loss.backward()

    optimizer_G.step()


    # 2. Train Discriminator

    optimizer_D.zero_grad()
    real_loss = adversarial_loss(discriminator(real_imgs, text_embeddings),
torch.ones(batch_size, 1))

    fake_loss = adversarial_loss(discriminator(gen_imgs.detach(), text_embeddings),
torch.zeros(batch_size, 1))

    d_loss = (real_loss + fake_loss) / 2

    d_loss.backward()

    optimizer_D.step()



    return g_loss.item(), d_loss.item()
```