Model architecture description and illustration, training procedure (hyperparameters, optimization details, etc.):

Model Architecture Description:

1. Generator (Decoder):

- **Purpose**: Takes a latent vector z from the latent space and produces an image.
- **Architecture**: A series of **transposed convolutions (nn.ConvTranspose2d)** to progressively upscale the latent vector to a 3-channel image of size 64x64.
    - **Layers**:
        1. Input: z with size `(latent_dim x 1 x 1)`
        2. **ConvTranspose2d**:
            - z → 4x4x512: Latent vector is expanded to a spatial size of 4x4 with 512 channels.
        3. **BatchNorm2d** + **ReLU**: Stabilizes training and adds non-linearity.
        4. **Progressive Upscaling**:
            - 512→256→128→64: Sequential transposed convolutions increase spatial resolution, halving the number of channels at each step.
        5. Output: 64x64x3: Final layer outputs a 3-channel RGB image, normalized by a **Tanh** activation function to map values to [−1,1][-1, 1].

2. Discriminator:

- **Purpose**: Distinguishes between real and generated (fake) images.
- **Architecture**: A series of **down-sampling convolutions (nn.Conv2d)** to extract hierarchical features, reducing the input image to a single value (real/fake probability).
    - **Layers**:
        1. Input: 3x64x64(RGB image).
        2. **Conv2d + LeakyReLU**:
            - Downsamples the spatial dimensions while doubling the number of feature channels at each step: 64x64x3→32x32x64→16x16x128→8x8x256→4x4x512

        3. Final Output: 1x1x1(Sigmoid activation): Outputs a scalar probability for real/fake classification.

        4. **BatchNorm2d** and **LeakyReLU (0.2)**: Improve training stability and ensure non-linearity.

Weights Initialization:

- **Generator** and **Discriminator** weights are initialized using a normal distribution:
    - Mean = 0.0, Standard deviation = 0.02.
    - For **BatchNorm**, weights are initialized to 1.0, and biases are set to 0.

Model Architecture Illustration:

Generator

Input: z(latent_dim×1×1)
↓ ConvTranspose2d + BatchNorm + ReLU
↓ ConvTranspose2d + BatchNorm + ReLU
↓ ConvTranspose2d + BatchNorm + ReLU
↓ ConvTranspose2d + BatchNorm + ReLU
↓ ConvTranspose2d + Tanh

Output: 3×64×64

Discriminator

Input: 3×64×64

↓ Conv2d + LeakyReLU
↓ Conv2d + BatchNorm + LeakyReLU
↓ Conv2d + BatchNorm + LeakyReLU
↓ Conv2d + BatchNorm + LeakyReLU
↓ Conv2d + Sigmoid
Output: 1 (real/fake probability)

Training Procedure:

1. Initialization

1. **Device Setup**: Use GPU if available for faster computation:
   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
2. **Hyperparameters**:
    - **Batch Size**: Number of images per training step. (Examples: 16, 32, 64 depending on configuration.)
    - **Latent Space Dimension (z)**: The dimensionality of the random noise vector used by the Generator. (Examples: 50, 100, 200 depending on configuration.)
    - **Image Size**: 64×64
    - **Learning Rate (η)**: 0.0002 for both the Generator and Discriminator.

    - **Optimizer Parameters**: Adam optimizer with:
        - $\beta_1=0.5$, $\beta_2=0.999$
    - **Epochs**: 80
3. **Loss Function**:
    - Use **Binary Cross-Entropy Loss (BCELoss)** for:
        - Discriminator: Real vs. Fake classification.
        - Generator: Fooling the Discriminator.
4. **Weights Initialization**:
    - Initialize model weights using a normal distribution:
        - Convolution weights: Mean = 0.0, Std = 0.02
        - Batch normalization weights: Mean = 1.0, Bias = 0.0

2. Training Loop

For each **epoch** and **batch**:

1. **Load Real Images**:
    - Load a batch of real images from the dataset and normalize them to the range $[-1,1]$.
2. **Train the Discriminator**:
    - **Inputs**:
        - Real images from the dataset.
        - Fake images generated by the Generator using a random noise vector $z \sim N(0,1)$.
    - **Targets**:
        - Real images: Labels y_{real}(label smoothing to improve stability).
        - Fake images: Labels $y_{fake}=0.0$
    - **Steps**:
        - Compute loss for real images: $L_{real}=BCELoss(D(real\_images),y_{real})$
        - Compute loss for fake images: $L_{fake}=BCELoss(D(fake\_images),y_{fake})$
        - Total Discriminator loss: $LD=L_{real}+L_{fake}$
        - Backpropagate and update Discriminator weights:
          optimizer_d.zero_grad()

          L_D.backward()

          optimizer_d.step()
3. **Train the Generator**:

    - **Input**: Latent vector $z \sim N(0,1)$
    - **Objective**: Fool the Discriminator into classifying fake images as real ($y_{real}=0.9$).
    - **Steps**:

        - Generate fake images from z.
        - Compute Generator loss:
          L_G=BCELoss(D(fake_images),y_real)
        - Backpropagate and update Generator weights:
          optimizer_g.zero_grad()

          L_G.backward()

          optimizer_g.step()

4. **Logging and Monitoring**:

    - Record **Generator loss (L_G)** and **Discriminator loss (L_D)** after every batch.
    - Print progress periodically (e.g., every 100 batches):

5. **Checkpointing**:

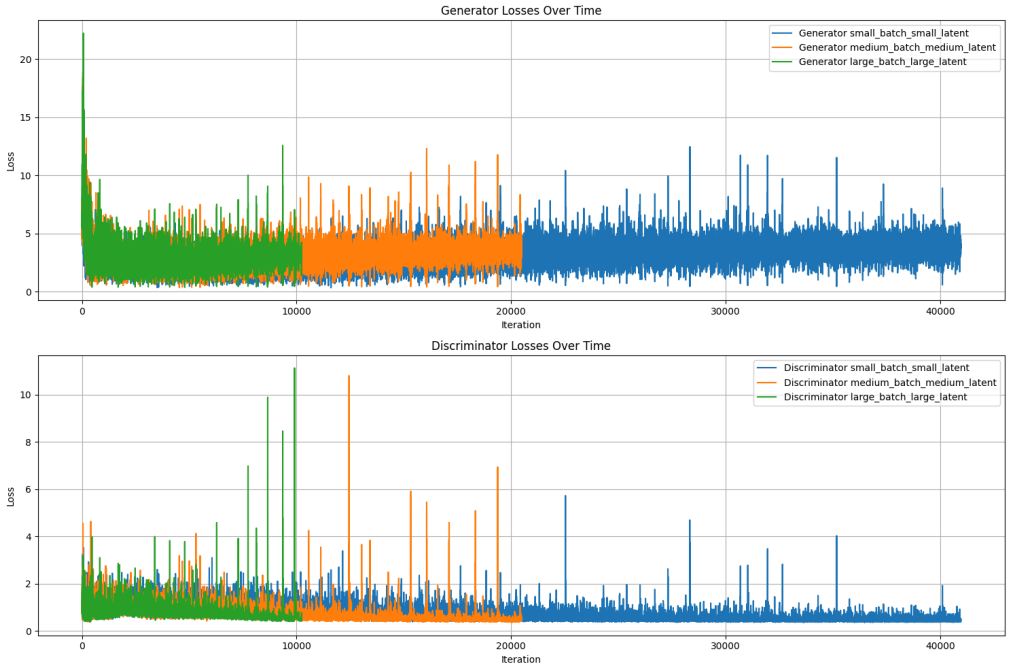
    - Save the following periodically (e.g., every 10 epochs):
        - **Generated Images**: Use a fixed latent vector z to generate and save sample images.
        - **Model Weights**: Save the Generator and Discriminator model states for later use.

3. Output

- **Generated Images**:
    - Images are saved periodically (every 10 epochs) for visual inspection.
- **Loss Curves**:
    - Plot and save the Generator and Discriminator loss curves over time for training evaluation.
- **Model Checkpoints**:
    - Save trained weights of the Generator and Discriminator at regular intervals.

Training convergence plots as a function of training time:

○ GAN: discriminator and generator losses:



The loss graphs show that smaller batch sizes and latent dimensions converge faster initially but show higher instability, while larger configurations provide smoother loss behavior but require more iterations to stabilize. The "medium_batch_medium_latent" configuration appears to have the best balance, with relatively stable losses and fewer extreme fluctuations. The periodic spikes in both generator and discriminator losses suggest training instability, which is common in GANs and may come from imbalanced learning dynamics, batch sizes, or learning rates.

Summary of your attempts and conclusions. Your conclusions and explanations should be based on the actual results you received during your attempts. Include 1-2 pages of visualizations (the images your model produces):

In this experiment, we trained multiple GAN configurations with varying batch sizes and latent dimensions to assess their impact on training stability and image quality.

The three configurations tested were: small batch size (16) with small latent dimension (50), medium batch size (32) with medium latent dimension (100), and large batch size (64) with large latent dimension (200). Throughout the training, we monitored the generator and discriminator losses and periodically saved generated images to evaluate qualitative performance.

The results indicate that smaller batch sizes and latent dimensions converged more quickly initially but displayed greater instability, as evidenced by significant fluctuations in the loss curves. These configurations produced images that lacked clear structure during early epochs and improved marginally over time. On the other hand, the larger batch and latent dimension configurations exhibited smoother loss behaviors, though they required more iterations to stabilize. These models generated more coherent and detailed images, especially during later epochs.

The "medium_batch_medium_latent" configuration provided the best balance between convergence speed and stability. Its losses were relatively stable, with fewer extreme fluctuations, and it generated visually plausible images earlier in the training process compared to the other configurations. The periodic spikes in the loss curves across all configurations highlight the inherent instability in GAN training, likely caused by imbalanced generator-discriminator dynamics.

Visualizations

The following images illustrate the progression of generated images across epochs for each configuration:

Batch size = 16, latent dimension = 50:

- After epoch 10: Noisy, low-quality images with minimal structure.
- After epoch 40: Improved structure but still significant artifacts.
- After epoch 80: Slightly clearer images but far from realistic.

Batch size = 32, latent dimension = 100:

- After epoch 10: Basic structure begins to emerge, though still noisy.
- After epoch 40: Noticeable improvement with more defined features.
- After epoch 80: Significantly better coherence and detail.

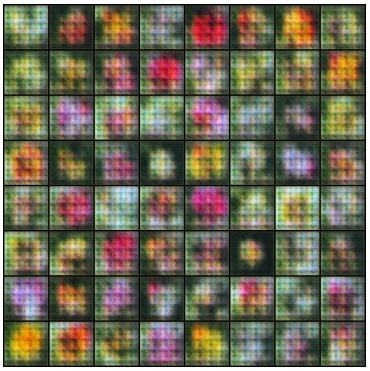
Batch size = 64, latent dimension = 200:

- After epoch 10: Highly noisy, initial attempts at structure.
- After epoch 40: Better feature representation and reduced noise.
- After epoch 80: The clearest and most coherent images among the configurations.

<u>IMAGES THE MODEL PRODUCES:</u>

Batch size = 16, latent dimension = 50:

After epoch 10:



After epoch 40:

After epoch 80:

Batch size = 32, latent dimension = 100:

After epoch 10:

After epoch 40:

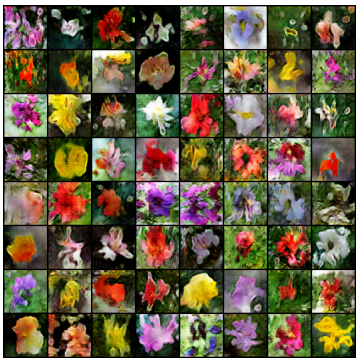
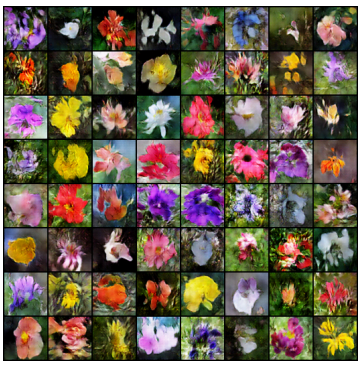
After epoch 80:

Batch size = 64, latent dimension = 200:

After epoch 10:

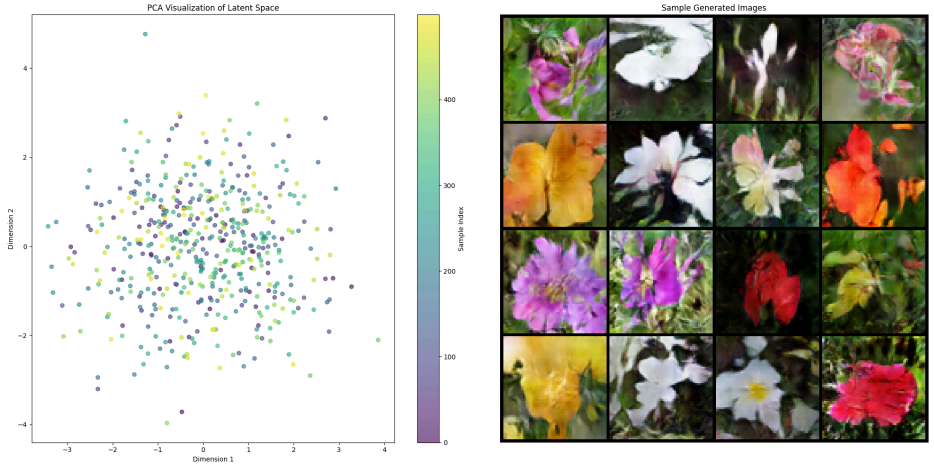
After epoch 40:

After epoch 80:

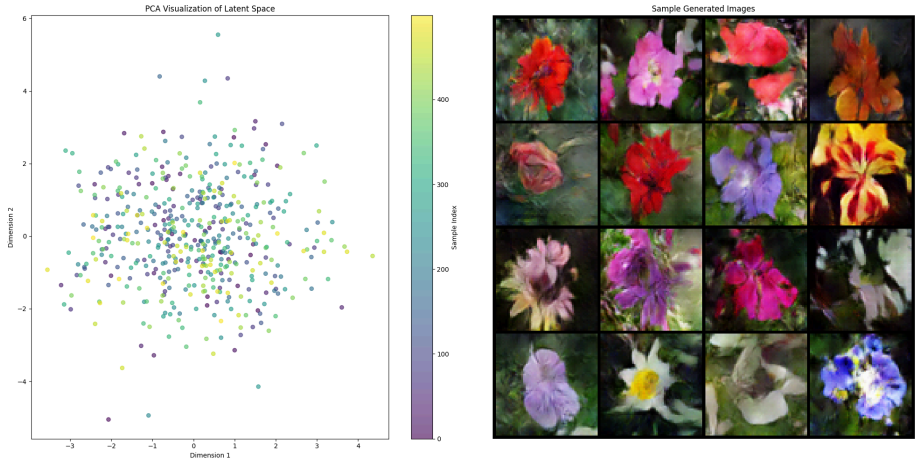




Latent spaces:

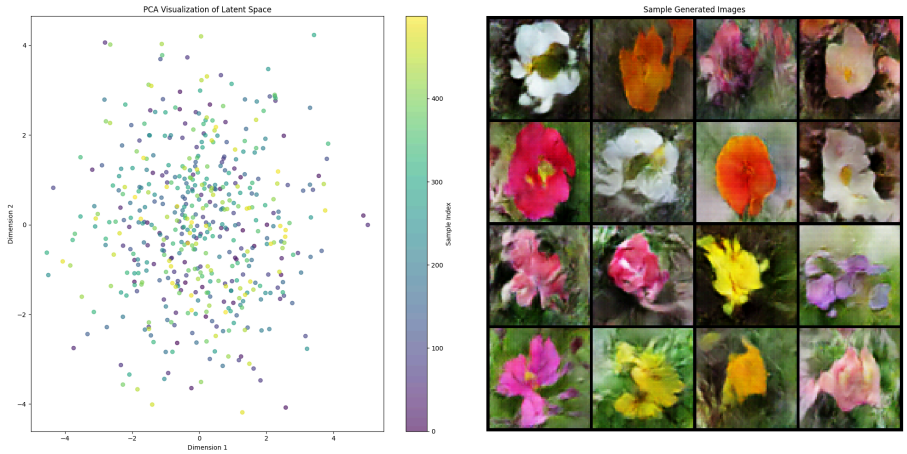
Batch size = 16, latent dimension = 50:



Batch size = 32, latent dimension = 100:



Batch size = 64, latent dimension = 200:



The small configuration (batch size 16, latent dim 50) shows a compact, gaussian distribution with decent but somewhat blurry image generation. The medium configuration (batch size 32, latent dim 100) appears to generate better image quality with more defined flower structures and a well-distributed latent space that maintains good coverage. However, the large configuration (batch size 64, latent dim 200) actually shows degraded performance - while the latent space becomes more structured with visible patterns in point distribution, the generated images show less definition, suggesting that the increased capacity may be making optimization more difficult without providing additional benefits.

While one might expect that larger batch sizes and latent dimensions would consistently improve results, the visualizations show that there's an optimal middle ground where the model has enough capacity to learn meaningful representations without becoming too difficult to train effectively.