UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Introduction to Deep Learning

Midterm project report on

Deep Convolutional GAN (DCGAN) for image super-resolution

Submitted by:

Cao Thái Hà	**(22BI13136)**
Hoàng Kỳ Duyên	**(22BI13128)**
Nguyễn Hoàng Duy	**(22BI13122)**
Nguyễn Anh Đức	**(22BI13089)**
Tạ Minh Đức	**(22BI13098)**
Phan Lạc Hưng	**(22BI13186)**

OCTOBER, 2024

Introduction

1. Problem Definition

- Images with low resolution frequently lack detail and clarity, which makes them unsuitable for applications that demand high-quality visuals. Our goal is to enhance the resolution and quality of these images.

2. Objective

- We propose a Deep Convolutional Generative Adversarial Network (DCGAN) method to enhance the quality of low-light images. This system comprises two parts: the generative network and the discriminative network.
- The objective of the generative network is to enhance low-light images in order to enhance the quality of the images.

Background

1. Image Super-Resolution Overview

- Image Super-Resolution is a machine learning task aimed at enhancing the resolution of an image while preserving its content and details.
- This technique is applicable in various domains, including computer vision and medical imaging, resulting in a high-resolution version of the original image.

2. Generative Adversarial Networks (GANs)

- A Generative Adversarial Network (GAN) is a deep learning framework that involves two neural networks competing against each other to create realistic new data from a given training set.
- One network generates new data by taking an input data sample and modifying it as much as possible.
- The other network tries to predict whether the generated data output belongs in the original dataset, essentially determining if the data is real or fake.

3. Deep Convolutional GAN (DCGAN)

- Deep Convolutional GAN (DCGAN) is a specific implementation of GANs that utilizes deep convolutional networks in both the generator and discriminator.
- The generator takes a random noise vector as input and uses fractional-strided convolutions to upsample it into a high-resolution image, applying ReLU activations in all layers except the output, which uses Tanh activation.
- The discriminator, on the other hand, uses strided convolutions to downsample the input image, employing LeakyReLU activations throughout.

- Both components utilize batch normalization to stabilize training.
- Key modifications from standard GANs include the replacement of fully connected layers with convolutional layers, the use of batch normalization, and specific activation functions, all of which contribute to more stable training and higher quality image generation.

Dataset and preprocessing

1. Dataset

- In this project, we used the CelebA dataset, which is a large-scale face attributes dataset containing over 200.000 celebrity images with various facial expressions, backgrounds, and lighting conditions. CelebA is widely used for image generation tasks, making it a suitable dataset for training a DCGAN model.

2. Preprocessing

- Before training the DCGAN model, the dataset passed through several steps to ensure the compatibility with the architecture:
- Resizing: All images were resize to 64x64 pixels, matching the input size expected by the model
- Normalization: Pixel values were normalized to the range [-1,1] by subtracting 0.5 and dividing by 0.5, which helps with stabilizing the training process in the generator and discriminator.
- We utilized PyTorch's dataloader for loading the dataset in 32 batches, setting shuffle to ensure that the batches of images are varied throughout training and using 2 workers for efficient loading and preprocessing of the dataset.

DCGAN Model

1. Generator Architecture

- The generator in our DCGAN takes a 100-dimensional random noise vector as input and outputs a 64x64 RGB image. It uses 5 transposed convolutional layers to upsample the input noise into an image, progressively increasing the spatial dimensions while reducing the number of feature maps. We also apply the batch normalization and ReLu activations through each layer to help with stability and gradient flow.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

2. Discriminator Architecture

- The discriminator in this CDGAN model takes a 3x64x64 image as input and output outputs a fully connected layer followed by a sigmoid function to produce the probability. It also uses 5 convolutional layers to downsample the input.

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

3. Hyperparameters

- Learning rate: 0.00002
- Batch size: 128
- Latent vector (z) Size: 100
- Optimizer: Adam optimizer with B1 = 0.5 and B2 = 0.999

4. Training setup

- Number of epoch: 10 epochs for the first experiment

- Hardware: In this project, we used Google Colab GPU for the training and using Pytorch's GPU support for faster calculation.
- Loss function: Binary Cross Entropy Loss (BCE Loss) was used for both generator and discriminator.

5. Training Procedure

a) Initialize model: The generator and discriminator were initialized, with their weights set using a normal distribution(N(0,0.2)).

b) Load data: The CelebA dataset was proprocessed, and the DataLoader was used to load mini_batches of images for each epoch.

c) Training loop:

- Train the discriminator
  - a batch of real images from the dataset was put into the discriminator, and the discriminator's real loss was computed
  - a batch of fake images was generated by feeding random noise into the generator, and the discriminator's fake loss was computed.
  - the discriminator's total loss was calculated, and backpropagation was performed to update its weights
- Train the generator
  - a new batch of fake images was generated
  - these fake images were passed through the discriminator again, and the generator's loss was computed based on how well it fooled the discriminator.
  - backpropagation was performed to update the generator's weights, with the goal of improving its ability to generate realistic images..

6. Output and Metrics:

- During training, metrics such as Discriminator Loss and Generator Loss were tracked to monitor convergence.
- Sample images were generated and saved at regular intervals to visually inspect the progress of the generator.

Experiments and Results

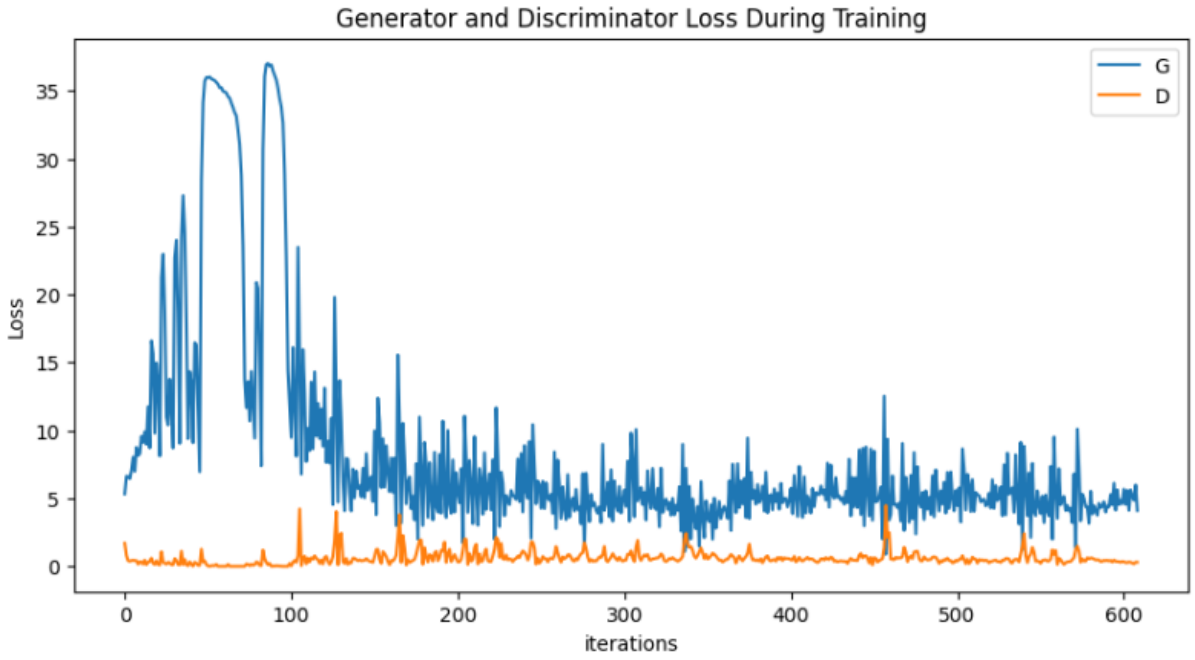
1. Experimental Setup

- The model was implemented using PyTorch and trained on Google Colab utilizing GPU acceleration for efficient computation. The primary libraries used include:
  - Pytorch
  - Torchvision
  - Matplotlib
  - Numpy

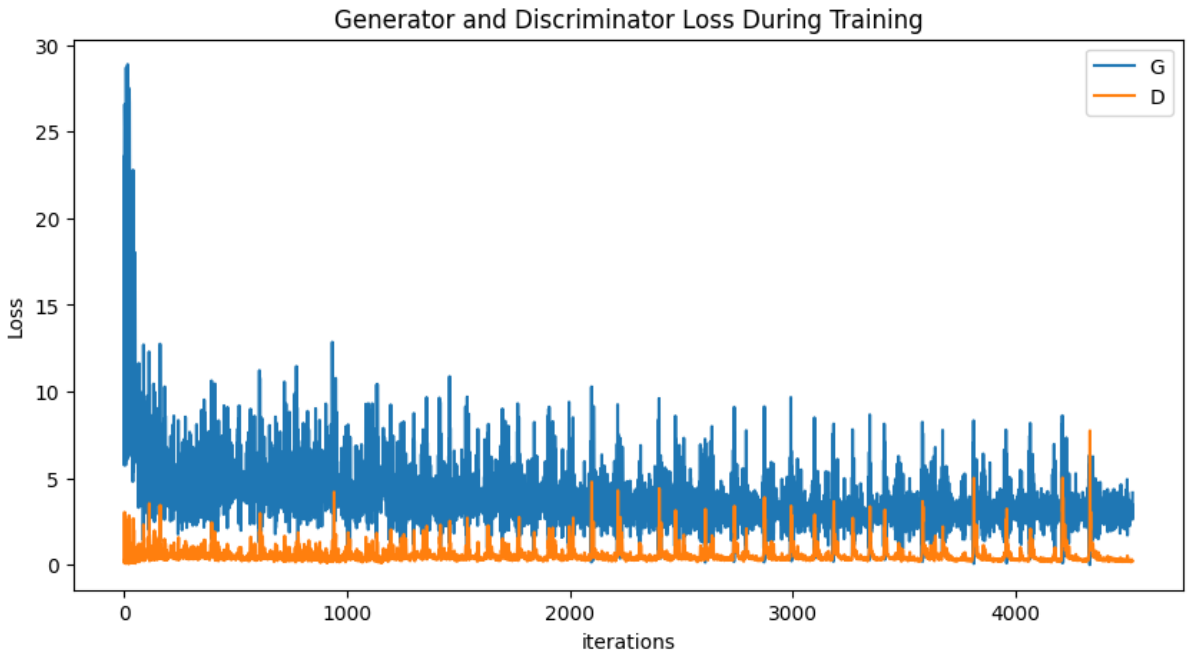
    - Google Drive API
- The model was trained using Adam optimizer with a learning rate of 0.0002, Beta1 set to 0.5, and batch size of 128. The training loop ran for 10 epochs, and images generated at intermediate stages were saved for qualitative analysis.

2. Results

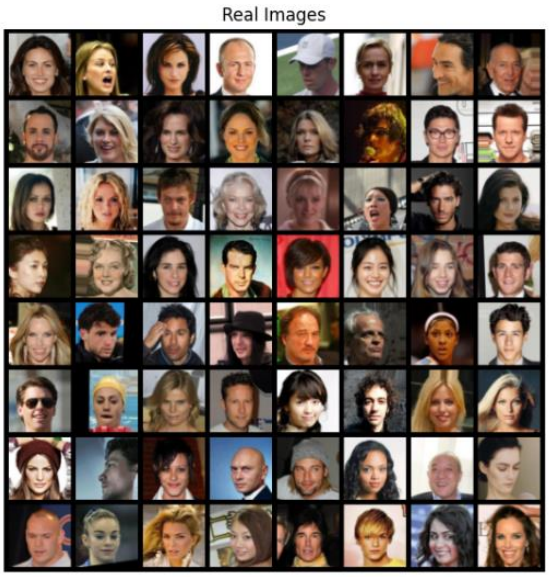
- In this experiment, we will look at the loss of Generator and Discriminator and see how it changed during the training.
- Iteration of training versus loss
    - After 10 epochs:



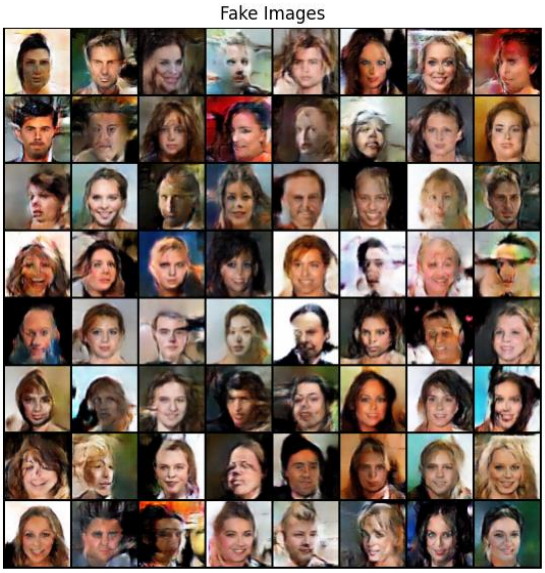
    - After 150 epochs:



- Finally, compare between real images and fake images
  - After 10 epochs:



  - After 150 epochs:



3. Discussion

- The trends observed from the loss chart show that the training is converging, with the generator improving its performance over time. The discriminator's loss remains low and stable, showing that the discriminator has adapted to the generator's output, indicating that it is still learning to keep pace.

4. Future Work

- Increase Dataset Diversity
- Modify the model to take different dataset
- Increase epoch to see better score
- Improve the Generator and Discriminator neural network
- Apply these GAN hacks to see the difference

References

- Large-scale CelebFaces Attributes (CelebA) Dataset
- DCGAN Tutorial
- https://www.kaggle.com/code/jesucristo/introducing-dcgan-dogs-images/notebook#Generated-Dogs
- https://www.kaggle.com/code/mayur7garg/dcgan-for-celebfaces/notebook
- https://nttuan8.com/bai-1-gioi-thieu-ve-gan/
- https://nttuan8.com/bai-2-deep-convolutional-gan-dcgan/#Cau_truc_mang
- https://pyimagesearch.com/2021/12/13/gan-training-challenges-dcgan-for-color-images/
- UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS