

**MINISTRY OF EDUCATION AND TRAINING
HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION
FACULTY OF MECHANICAL ENGINEERING**



**PROJECT REPORT
FACIAL IMAGE GENERATION APPLICATION
WITH DCGAN MODEL**

Group: 7

Students: Nguyen Dang Duy Tan20134007

Tong Huynh Tanh.....20134024

Tran Van Thoai.....20134025

Lecturer: PhD. Tran Nhat Quang

Ho Chi Minh City, May 2023

CONTENTS

ACKNOWLEDGEMENT	1
UNDERSTANDING PLAGIARISM	2
COMMITMENT	3
CHAPTER 1: OVERVIEW	4
1.1. Rational and reason for choosing the topic	4
1.2. Aim of the study	4
1.3. Limitation	5
CHAPTER 2: THEORICAL FOUNDATION	6
2.1. Introduction to GAN (Generative Adversarial Network).....	6
2.2. GAN Application.....	8
2.2.1. Generate Realistic Photographs.....	8
2.2.2. Image-to-Image Translation	8
2.2.3. Text-to-Image Translation (text2image)	10
2.3. GAN Architecture.	12
2.4. DCGAN: Deep Convolutional Generative Adversarial Network	13
2.5. Generator	14
2.6. Discriminator	17
2.7. Loss.....	17
2.7.1. Define Discriminator loss.....	17
2.7.2. Define Generator loss	18
CHAPTER 3: TRAINING DCGAN	19
3.1. Introduction to face dataset	19
3.2. Preprocessing the dataset.....	19

3.3. Training process/ Code explanation	22
3.3.1. Import the libraries	22
3.3.2. Prepare the dataset.....	23
3.3.3. Making the GENERATOR and DISCRIMINATOR function.....	24
3.3.4. Define the DCGAN model	24
3.3.5. Training and validating the model.....	26
3.3.6. Monitoring the process	28
3.3.7. Declare the parameters for the model and start the training process.	30
3.3.8. Training result	30
CHAPTER 4: CONCLUSION AND FUTURE WORK	33
4.1. Conclusion	33
4.2. Future work.....	33
REFERENCE	34

ACKNOWLEDGEMENT

We have completed three-quarters of the school's training program after three years of hard effort and acquiring specialized knowledge. Throughout the journey, we have received dedicated instruction from the teachers of the University of Technology and Education of Ho Chi Minh City, as well as the teachers of the Faculty of Mechanical Engineering. That useful knowledge was essential in completing this project. As a result, we'd like to start by thanking all the teachers in the Ho Chi Minh City University of Technology and Education in general, and the Mechanical Engineering faculty in particular.

We would like to express our gratitude to teacher Tran Nhat Quang. The teacher has dedicatedly guided us to complete the project in the best way possible; without his guidance, this project may not be completed.

However, mistakes cannot be prevented when doing so. To make this report more thorough, we welcome your comments and suggestions.

Finally, we would like to express our gratitude to our teacher once again.

Sincerely,

Nguyen Dang Duy Tan

Tong Huynh Tanh

Tran Van Thoai

UNDERSTANDING PLAGIARISM

Definition:

Plagiarism is a serious ethical and academic offence where someone presents another person's thoughts, words, or work as their own without giving due credit to the actual author. The qualities of integrity, honesty, and originality are undermined by this behavior, which is seen as intellectual theft and lying.

Type of plagiarism:

- Directly copying someone else's work without proper credit or citation. Paraphrasing someone else's ideas or information without acknowledging the original source.
- Patchwriting, which involves rewriting someone else's work with minor changes while keeping the overall structure and content intact, without giving proper attribution.
- Self-plagiarism, which is reusing one's own previous work without indicating it as such.
- Collusion, where individuals collaborate and present someone else's work as their own without proper acknowledgment.
- Citation errors or inappropriate referencing, including incorrect or missing citations or references.

It is crucial to understand and avoid all forms of plagiarism by correctly attributing and citing sources in academic or professional work.

COMMITMENT

We, the team members listed below, sincerely confirm that we alone completed this project. We guarantee that none of the documents, source code, or other materials used by us were taken without providing proper credit to the original author. We fully acknowledge our responsibility for any plagiarism offenses that may occur.

Member:

Nguyen Dang Duy Tan

Tong Huynh Tanh

Tran Van Thoai

CHAPTER 1: OVERVIEW

1.1. Rational and reason for choosing the topic

Industry 4.0, often known as the 4.0 revolution, marks in a new age of technical growth molded by artificial intelligence (AI). AI is gradually making its way into many human disciplines. AI can do actions that would ordinarily need human intellect, making it an effective tool for automating and optimizing a wide range of processes.

The ability of computers to "learn" on their own, understand and evaluate new data without human involvement is a characteristic of AI. Furthermore, AI's capacity to quickly handle massive volumes of data boosts its efficiency, making it well-suited for a variety of applications.

To keep up with the current trend of the Industry 4.0 revolution, we have decided to choose the topic of developing a facial image generation application. This subject not only helps our team gain a better understanding and further practice in AI, but it also serves as our final project for this subject. At the moment, the field of generative applications has enormous potential. It has the ability to produce innovative and inventive pictures in art and design by combining distinct qualities from many sources, resulting in one-of-a-kind and ground-breaking artwork. It can also sharpen photos, translate text to images, and do a variety of additional tasks.

Our goal in building a face image creation application based on the GAN (Generative Adversarial Network) model is to contribute to the growth of this discipline by creating more refined, detailed, and effective generation systems.

1.2. Aim of the study

- Determine the most suitable model for the dataset, giving out the best accuracy.
- Understand the basic components of machine learning models.
- Data selection and preprocessing.
- Training the model and make application to apply.

1.3. Limitation

- It took a lot of time to train the model.
- They take a lot of memory to execute.
- The accuracy might high but image generated is not good enough

CHAPTER 2: THEORETICAL FOUNDATION

2.1. Introduction to GAN (Generative Adversarial Network)

GANs, or Generative Adversarial Networks, are a generative modeling approach that employs deep learning approaches involving convolutional neural networks. [1] It aims to generate new data samples that are similar to a given training dataset.



Fig.2.1. GAN examples for generating new plausible examples for image datasets

Within the domain of artificial intelligence and machine learning, it has emerged as a transformative innovation. GANs, initially introduced by Ian Goodfellow and his team in 2014, have captivated the interest of numerous researchers and have quickly become a thrilling field of exploration within deep learning research.

GANs typically operate in an unsupervised mode and utilize a framework where one party's gain corresponds to the other party's loss just like the Minimax algorithm in chess or in any two-player competitive game.

GAN is composed of two models: the generator (G) and the discriminator (D). To explain the model in an easy-to-understand way, we can imagine a crime game involving a criminal and a police officer as follows:

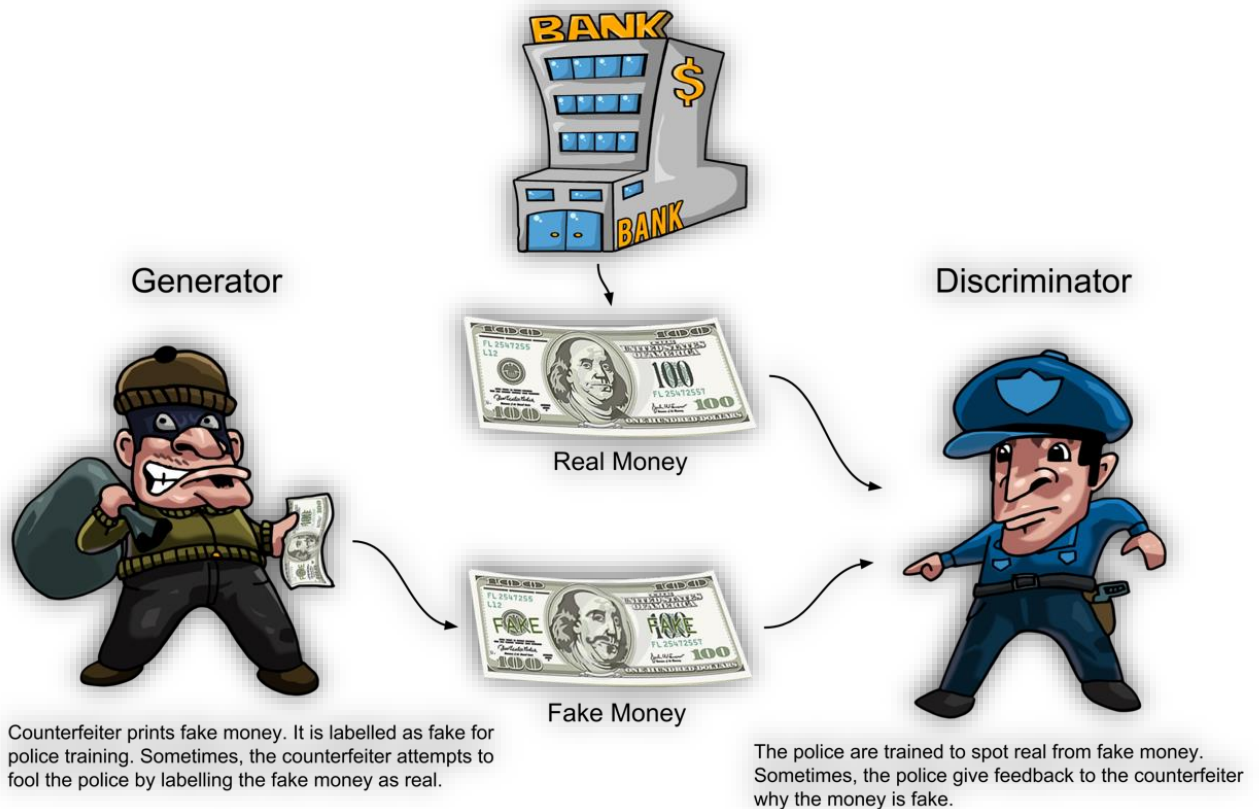


Fig.2.2. GAN generators and discriminators are equivalent to counterfeiters and cops. The counterfeiter's purpose is to trick the police into thinking the dollar bill is real.

The criminal (G) creates counterfeit money, while the police officer (D) learns how to distinguish between real and fake currency. As the police officer tries to differentiate between real and fake money, the criminal relies on the feedback from the police officer to improve their ability to create counterfeit money, attempting to deceive the officer into misidentifying it [2]. The competition in this game promotes both teams to improve their methods until the fakes are indistinguishable from the real ones.

2.2. GAN Application

2.2.1. Generate Realistic Photographs

Andrew Brock et al. demonstrate the development of synthetic photos that are nearly indistinguishable from actual photographs in their 2018 study titled "Large Scale GAN Training for High Fidelity Natural Image Synthesis." [5]



Fig.2.3. Example of Realistic Synthetic Photographs Generated with BigGAN Taken from Large Scale GAN Training for High Fidelity Natural Image Synthesis, 2018.

2.2.2. Image-to-Image Translation

In their 2016 publication titled "Image-to-Image Translation with Conditional Adversarial Networks," Phillip Isola et al. show GANs, notably their pix2pix technique, for several image-to-image translation applications.

Examples include translation tasks such as:

- Translation of semantic images to photographs of cityscapes and buildings.
- Translation of satellite photographs to Google Maps.
- Translation of photos from day to night.
- Translation of black and white photographs to color.

- Translation of sketches to color photographs.



Fig.2.4. Example of Photographs of Daytime Cityscapes to Nighttime With pix2pix. Taken from Image-to-Image Translation with Conditional Adversarial Networks, 2016.



Fig.2.5. Example of Sketches to Color Photographs With pix2pix. Taken from Image-to-Image Translation with Conditional Adversarial Networks, 2016.

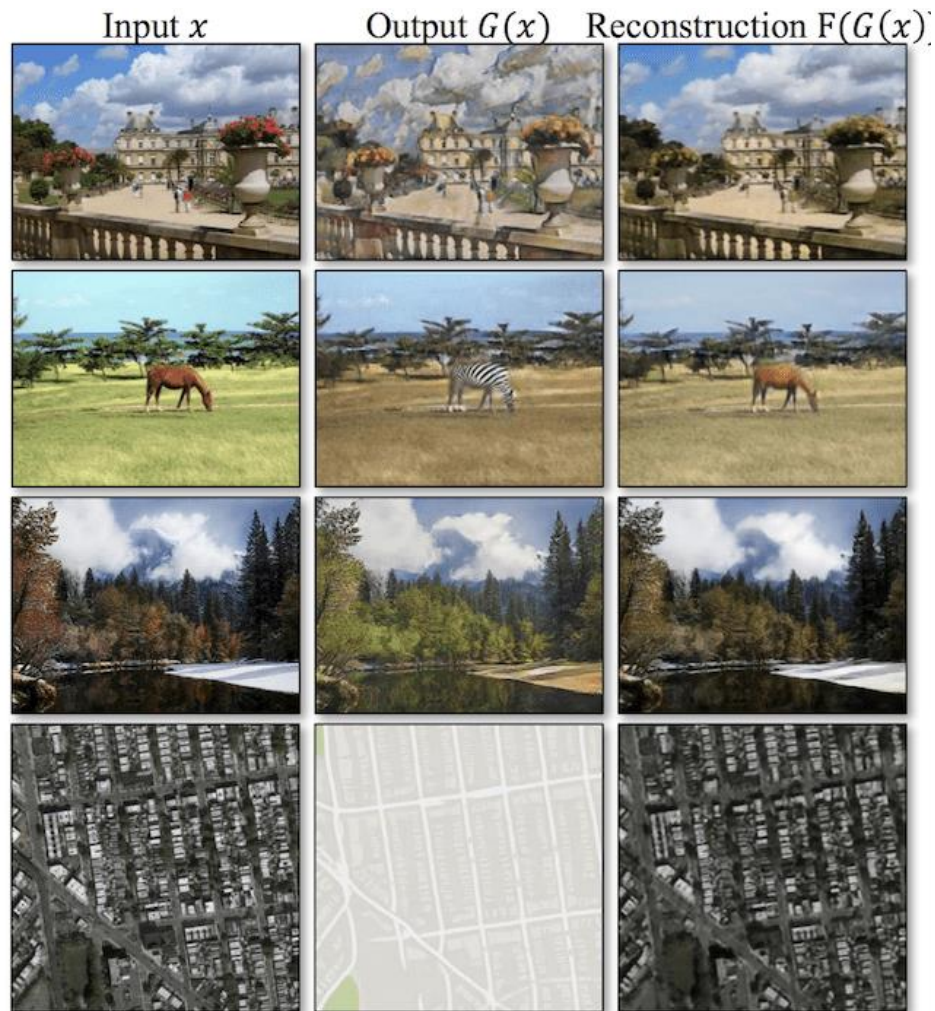


Fig.2.6. Example of Four Image-to-Image Translations Performed With CycleGANTaken from Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017.

2.2.3. Text-to-Image Translation (text2image)

Han Zhang et al. demonstrate the use of GANs, specifically their StackGAN, to generate realistic looking photographs from textual descriptions of simple objects such as birds and flowers in their 2016 paper titled "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks.".

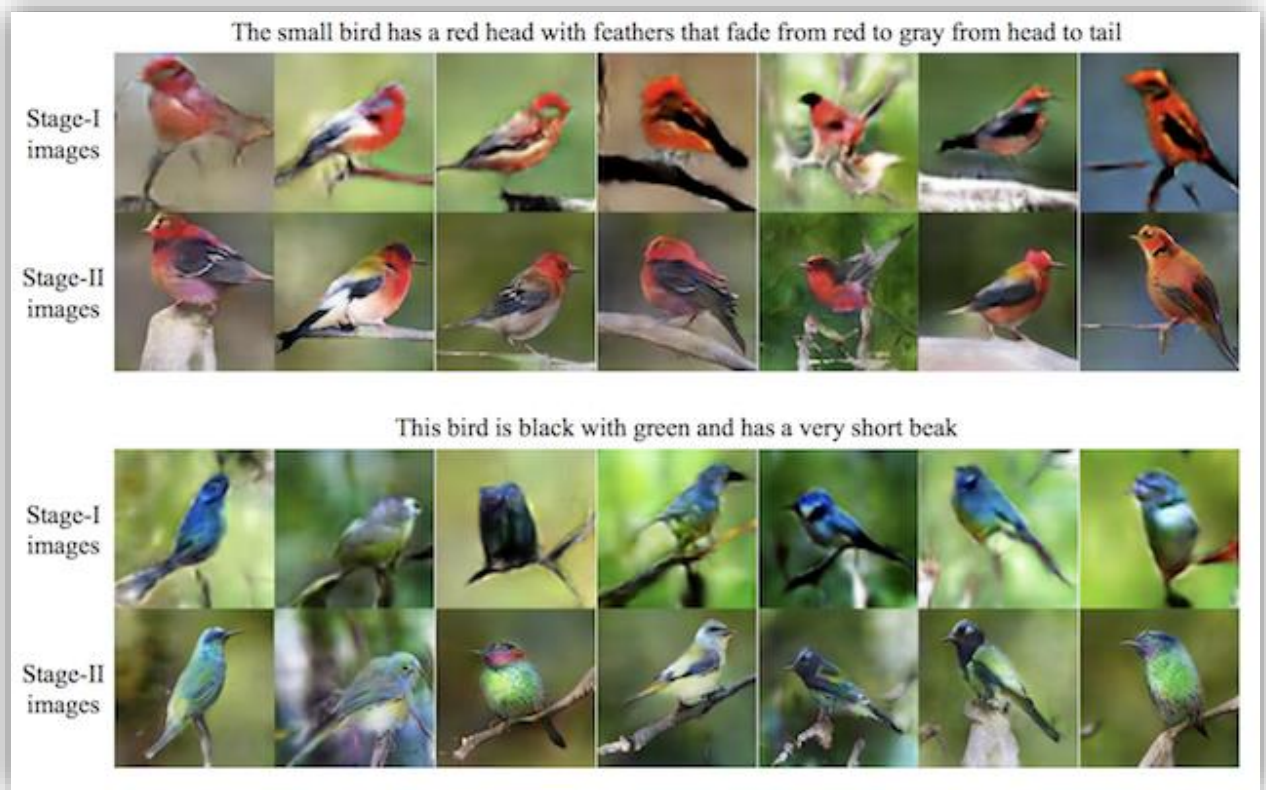


Fig.2.7. Example of Textual Descriptions and GAN-Generated Photographs of Birds Taken from StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, 2016.

***More and more application**

GANs (Generative Adversarial Networks) have found several applications in a variety of disciplines, demonstrating their adaptability and promise. Several important GAN applications have been explored in this area. However, with the rapid development of this subject, countless new applications are predicted to arise, illustrating the constantly growing variety of GAN application potential.

2.3. GAN Architecture.

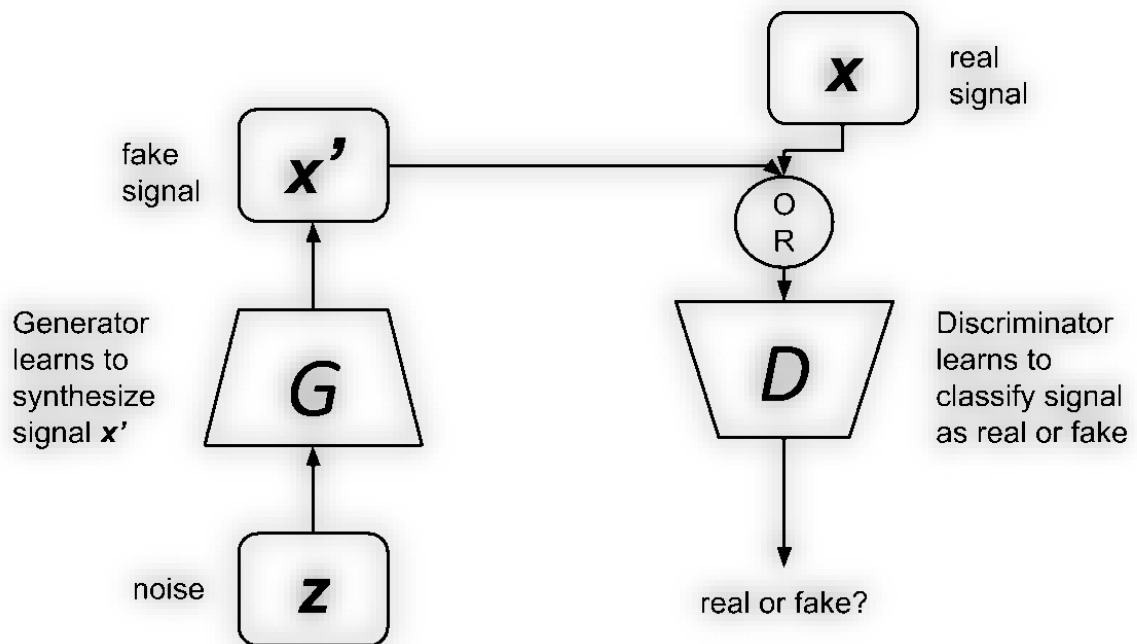


Fig.2.8. The discriminator has been taught to tell the difference between real and fake signals or data. The generator learns to produce fake signals or data in order to trick the discriminator.

Figure 2.3 shows that a GAN model consists of two networks: a generator and a discriminator. The generator use noise as input and produces a synthesized signal. In contrast, the discriminator receives either a real or a synthesized signal as input. Real signals are obtained from actual sampled data, while fake signals are generated by the generator. Real signals are labeled as 1.0, indicating they are 100% likely to be real, while synthesized signals are labeled as 0.0, indicating they have 0% probability of being real. Since the labeling process is automated, GAN is still categorized as part of unsupervised learning in deep learning.

The discriminator's primary goal is to learn from the supplied dataset how to distinguish between true and fake signals. Only the discriminator's parameters are modified during the GAN training process. The discriminator, like a normal binary classifier, is trained to predict confidence values ranging from 0.0 to 1.0, reflecting the closeness of a particular input signal to a true one. This, however, is only one part of the overall process.

The generator periodically pretends that its output is a real signal and asks the GAN to categorize it as 1.0. When the discriminator receives the fake signal, it naturally classifies it as fake, assigning a label close to 0.0. The optimizer calculates updates to the generator's parameters based on the presented label (1.0) and the discriminator's prediction, incorporating this new training data. Essentially, the discriminator exhibits some level of uncertainty regarding its prediction, and the GAN takes this uncertainty into account. At this stage, the gradients propagate backwards from the discriminator's last layer to the generator's first layer. However, it is common practice to temporarily freeze the discriminator's parameters during this training phase. The generator employs these gradients to update its parameters and enhance its ability to synthesize fake signals [3].

The popular architecture of GAN is DCGAN (Deep Convolutional GAN), where both the generator (G) and discriminator (D) are deep convolutional networks, as depicted in the diagram below:

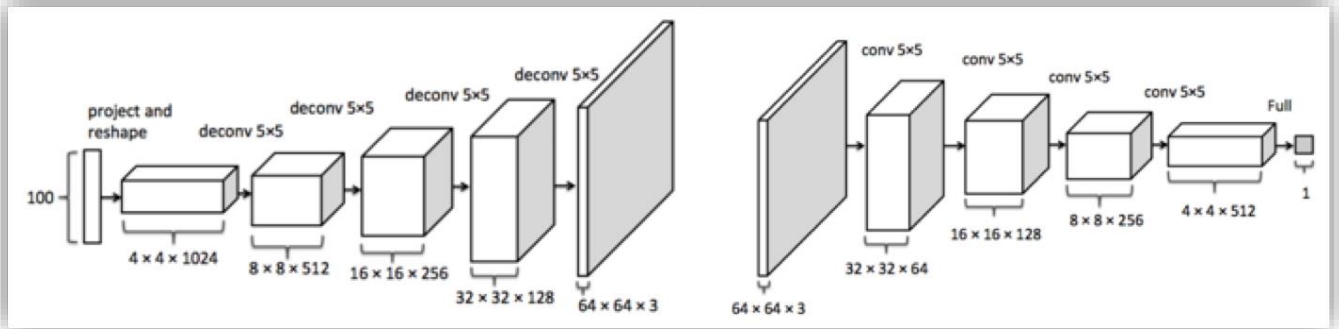


Fig.2.9. GAN Architecture

2.4. DCGAN: Deep Convolutional Generative Adversarial Network

A DCGAN is an extension of the previously mentioned GAN that specifically incorporates convolutional and convolutional-transpose layers in the generator and discriminator, respectively. This architecture was initially introduced by Radford et. al. in the paper "Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks." The discriminator consists of layers of strided convolutions, batch normalization, and LeakyReLU activations. It takes a 3x64x64 input image and produces a probability value indicating whether the input is real or fake. On the other hand, the

generator is composed of convolutional-transpose layers, batch normalization, and ReLU activations. It takes a latent vector, denoted as 'z', which is sampled from a standard normal distribution, and generates a 3x64x64 RGB image. The strided conv-transpose layers enable the transformation of the latent vector into a volume with the same dimensions as an image.[6]

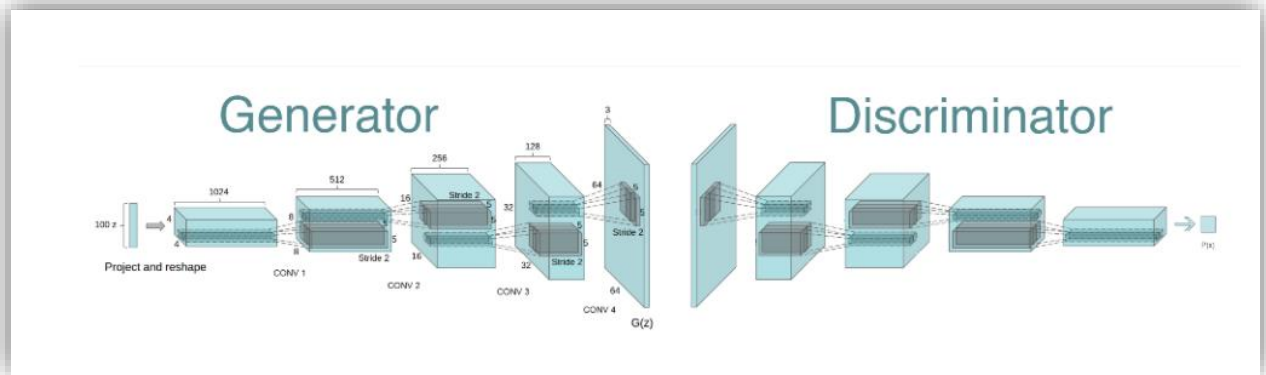


Fig.2.10. DCGAN Architecture

2.5. Generator

Generator is part of the model architecture used to generate new data. The generator's main task is to take as input a random noise vector (often called a latent space) and generate synthetic data samples similar to the training data.

Generator input is noise because it allows to create variety and distortion in the data generation process. For example, a person's face can have many variations but still be recognized as a human face. By using noise as input, Generator is capable of generating different variations of the face as we change random noise. This allows the Generator to create new, diverse and unique faces that resemble natural variations in actual faces.

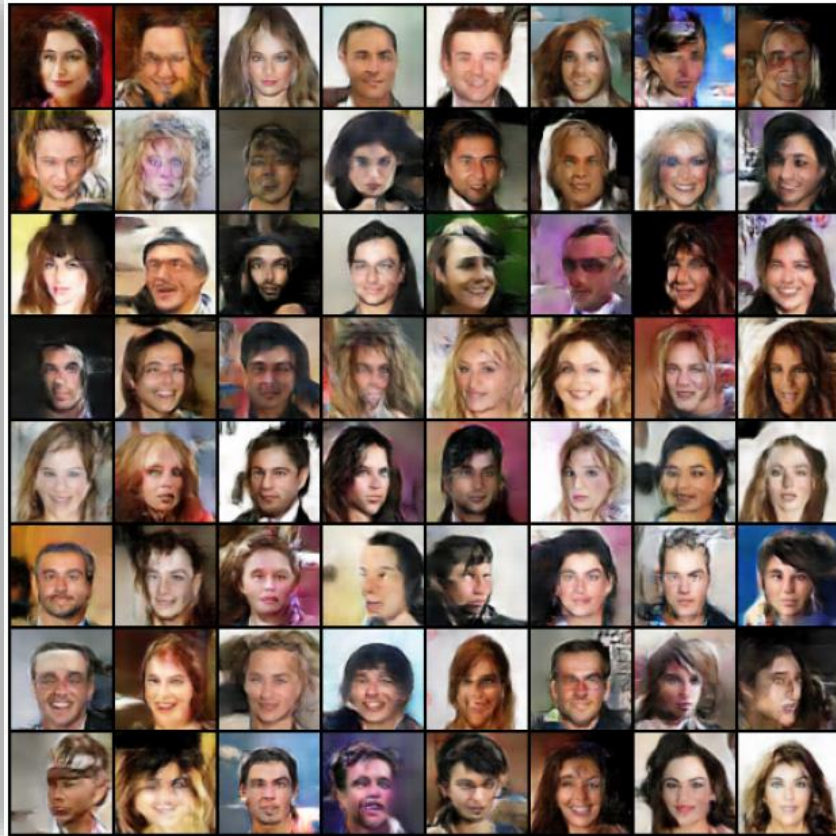


Fig.2.11. The generated faces differ from each other based on different noise inputs

In this project, we will use the generator architecture like below:

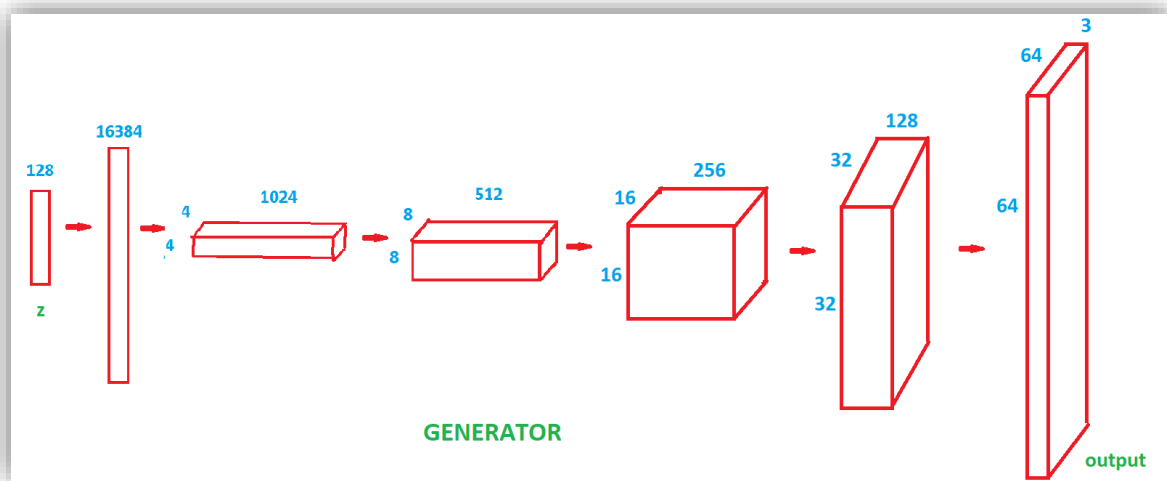


Fig.2.12. Generator architecture

The layers in the network are as follows:

- Dense (fully-connected) layer: $128 \times 1 \rightarrow 16384$
- Reshape layer: Converts the vector to a 3D tensor: $16384 \rightarrow 4 \times 4 \times 1024$
- Transposed convolution layer with stride=2 and kernel=256: $4 \times 4 \times 1024 \rightarrow 8 \times 8 \times 512$
- Transposed convolution layer with stride=2 and kernel=128: $8 \times 8 \times 512 \rightarrow 16 \times 16 \times 256$
- Transposed convolution layer with stride=2 and kernel=64: $16 \times 16 \times 256 \rightarrow 32 \times 32 \times 128$
- Transposed convolution layer with stride=2 and kernel=3: $32 \times 32 \times 128 \rightarrow 64 \times 64 \times 3$

First, the input noise (128) is passed through a fully connected layer to convert it to 16384 ($= 4 \times 4 \times 1024$). The number 16384 is chosen to reshape it into a 3D tensor ($4 \times 4 \times 1024$). Then, a transposed convolution with a stride of 2 is used to gradually increase the size of the tensor to $8 \times 8 \rightarrow 16 \times 16 \rightarrow 32 \times 32 \rightarrow 64 \times 64$. When the tensor size reaches 64×64 (matching the width and height of the desire image in the dataset) resize it into 3 channels of RGB. Finally, we got the output shape (64,64,3).

* Transposed convolutional:

Transposed convolution, also known as deconvolution, can be considered as the reverse operation of convolution. While convolution with a stride > 1 helps reduce the size of an image, transposed convolution with a stride > 1 increases the size of the image. For example, using a stride of 2 and 'SAME' padding will double the width and height dimensions of the image.[4]

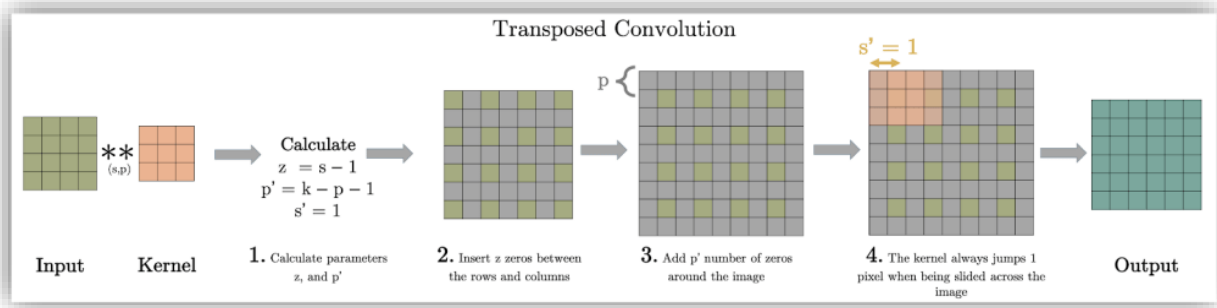


Figure2.13. Steps to perform transposed convolution

2.6. Discriminator

The discriminator (or discriminator network) is a component of the GAN architecture that is used to distinguish between real data samples from the training set and generated data samples created by the generator. So, the discriminator takes images as input and its output indicates whether the input image is real or fake.[2]

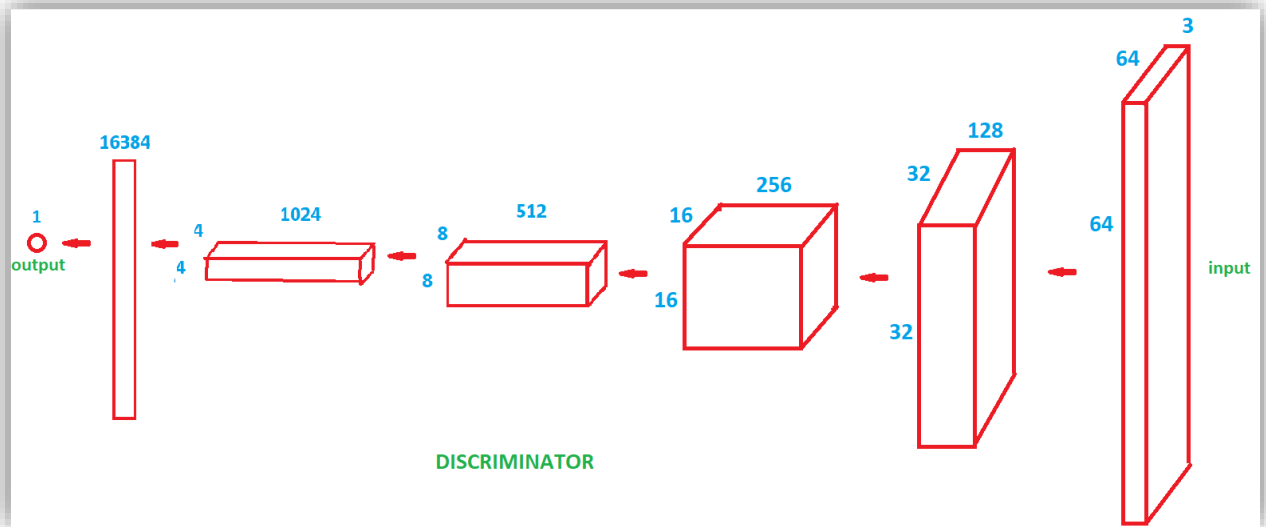


Fig.2.14. Discriminator architecture

The discriminator model is symmetrical to the generator model. The input image goes through convolutions with a stride of 2 to decrease the image size from 64x64 to 32x32, then to 16x16, 8x8, and finally to 4x4. As the size decreases, the depth of the tensor increases. Ultimately, the tensor with a shape of 4x4x1024 is reshaped into a -dimensional vector and passed through a fully connected layer to get the output.

2.7. Loss

2.7.1. Define Discriminator loss

In this task, the Discriminator has to distinguish between the real and fake image. Therefore, the Discriminator loss will be calculated by the difference between the Discriminator output for real images and the images generated by the Generator. Since this is a simple classifier task, we will calculate the loss based on the binary-cross entropy function:

$$L_{BCE} = -\frac{1}{n} \sum_{i=1}^n (Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

In which, Y_i is the true label of the sample,

\hat{Y}_i is the model's predicted probability for the positive class (usually passed through a sigmoid function to ensure it falls within the range $[0, 1]$).

This formula calculates the loss based on the logarithm of the predicted probability and the true label. When Y_i is 1 (true), the loss corresponds to the logarithm of the predicted positive probability (\hat{Y}_i). When Y_i is 0 (false), the loss corresponds to the logarithm of the predicted negative probability ($1 - \hat{Y}_i$).

2.7.2. Define Generator loss

The Generator's performance is evaluated based on its ability to create images that closely resemble real images and fool the Discriminator. Similar to the equation of binary-cross entropy, but in this case, Y is set to 1 because we want the Generator to create images that the Discriminator classifies as real. \hat{Y} is the probability predicted by the Generator for the positive class (usually passed through a sigmoid function to ensure it falls within the range $[0, 1]$). It represents the Generator's estimate of the probability that the generated fake images are real.

CHAPTER 3: TRAINING DCGAN

3.1. Introduction to face dataset

The Human Faces dataset obtained from Kaggle (link: <https://www.kaggle.com/datasets/garidziracrispen/human-faces/code>). This dataset is a valuable resource for studying machine learning with many application in facial recognition, computer vision, and deep learning algorithms.

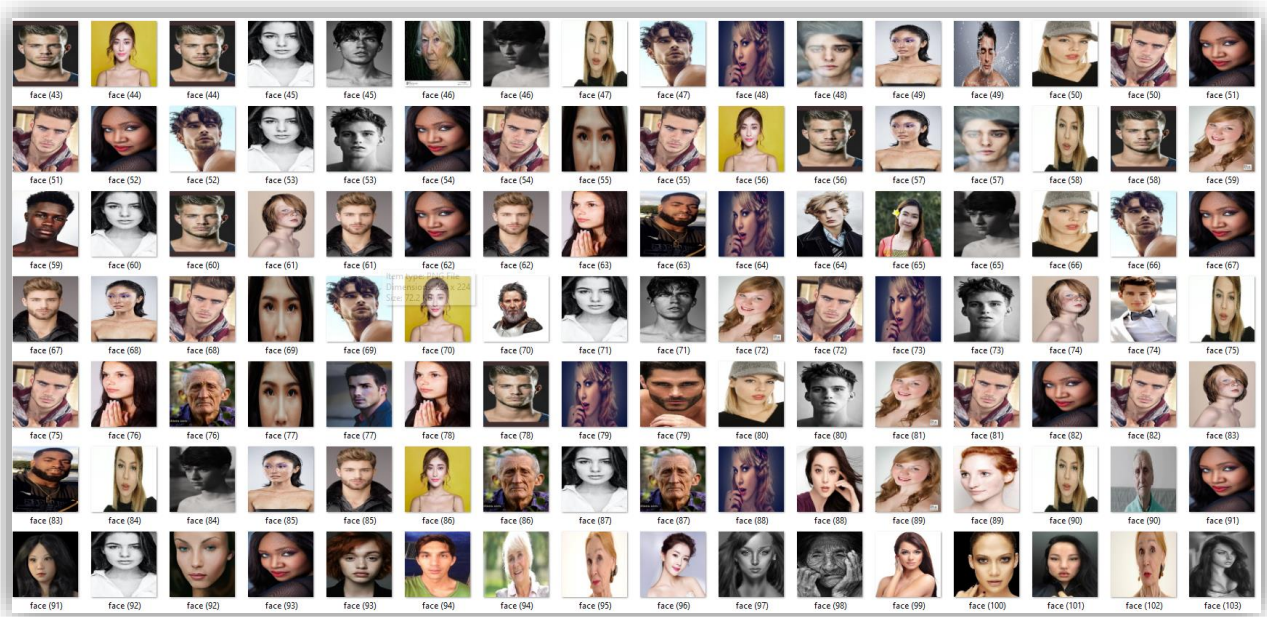


Fig.3.1. Face samples in the dataset

This data set includes 12000 photos with a variety of people from different ages, ethnicities to color colors, creating diversity for the data set. This dataset has many different shape and formats such as .jpg, .jpeg, .jfif, .png.

3.2. Preprocessing the dataset

To ensure consistency, we will convert all the images in the dataset to the jpg format, regardless of their original formats.


```

from PIL import Image
import os
import shutil
folder = r'Images'
count = 0
for file_name in os.listdir(folder):
    if file_name.endswith(".jpg"):
        file_path = folder + "/" + file_name
        shutil.copy(file_path, "convert_all")
        count += 1
    elif file_name.endswith(".jpeg"):
        img_path = folder + "/" + file_name
        im = Image.open(img_path)
        rgb_im = im.convert("RGB")
        rgb_im.save("convert_all/" + "face_" + str(count) + ".jpg" )
        count +=1
    elif file_name.endswith(".png"):
        img_path = folder + "/" + file_name
        im = Image.open(img_path)
        rgb_im = im.convert("RGB")
        rgb_im.save("convert_all/" + "face_" + str(count) + ".jpg" )
        count +=1

```

Fig.3.2. Python code to convert all format

The next step involves extracting faces from each image. This is necessary because the generator will need a clear background to generate realistic faces. Without this step, the generated results may be distorted or may contain unwanted artifacts from the background of original image. We will extract face by using a pre-trained MTCNN (Multi-Task Cascaded Convolutional Networks) model. MTCNN is particularly built for face identification and alignment, allowing us to recognize and extract facial areas from pictures with high accuracy. We can ensure that the Generator receives high-quality and well-aligned face as input, resulting in more realistic and visually appealing produced faces.

```

for filename in os.listdir(input_dir):
    if filename.endswith('.jpg') :
        input_path = os.path.join(input_dir, filename)

        # Đọc ảnh từ tệp tin
        image = cv2.imread(input_path)

        # Chuyển đổi từ BGR sang RGB
        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Sử dụng MTCNN để nhận dạng và cắt khuôn mặt
        boxes, _ = mtcnn.detect(rgb_image)
        if boxes is not None:
            for box in boxes:
                x1, y1, x2, y2 = box.astype(int)
                face = image[y1:y2, x1:x2]

                # Kiểm tra khuôn mặt không trống
                if face.size != 0:
                    # Xác định đường dẫn tệp tin đầu ra
                    output_filename = f"{filename}"
                    output_path = os.path.join(output_dir, output_filename)

                    # Lưu khuôn mặt vào tệp tin đầu ra
                    cv2.imwrite(output_path, face)

```

Fig.3.3. Python code for extract face

Finally, we will resize all of the photos to fit the model's input size. This step is required to verify that the photos have uniform dimensions and can be processed by the model correctly.


```

# resize và lưu lại ảnh
folder = r"pre_img_rename"
from PIL import Image
import os
for file_name in os.listdir(folder):
    image = Image.open(folder+"/"+file_name)
    width, height = image.size
    if width >= 64 and height >= 64:
        new_image = image.resize((128,128))
        new_image.save("pre_img_128x128"+ '/' + file_name)

```

Fig.3.4. Code for resize image



Fig.3.5. Result after pre-processing

3.3. Training process/ Code explanation

This is the most important step in this project. The training process was executed on Google Collab platform, which have a strong GPU acceleration and big storage.

The training step can be summarized as follow:

3.3.1. Import the libraries

This project is primarily built with Tensorflow, a well-known machine learning library. However, there are also extra common libraries used, such as NumPy, Matplotlib, ...etc.

```

import glob
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import os
from PIL import Image
from tensorflow import keras
from tensorflow.keras.preprocessing import image
from tensorflow.keras.layers import Input, Reshape, Dropout, Dense
from tensorflow.keras.layers import Flatten, BatchNormalization
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import LeakyReLU, ReLU, PReLU
from tensorflow.keras.layers import Conv2D, Conv2DTranspose
from tensorflow.keras.models import Sequential, Model, load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import Callback
from tensorflow.keras.preprocessing.image import img_to_array, load_img
from skimage.transform import resize
from scipy import linalg
import tensorflow as tf
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.applications.inception_v3 import preprocess_input

```

Fig.3.6. Import the libraries

3.3.2. Prepare the dataset

After pre-processing, the dataset is saved to Google Drive, we can get the images using these commands to synchronize:

```

# Sync google drive storage to this notebook.
from google.colab import drive
drive.mount('/content/drive')

```

Fig.3.7. Sync the dataset to the notebook

The dataset can then be unzipped (if necessary) and saved to a directory. Next, write a function to read the images in the dataset folder and save them as an array.

```
def read_images(data_dir):
    """
    This function reads all of the images in the specified directory and returns a Numpy array of the images.
    Arguments:
        data_dir: The directory containing the images.
    Returns:
        A NumPy array of the images, normalized to the range [0, 1].
    """
    # Create a list to store the images.
    images = []
    # Iterate over all of the files in the directory.
    for file_name in glob.glob(data_dir + '/*.jpg'):
        # Load the image from the file.
        img = image.load_img(file_name)
        # Convert the image to a NumPy array.
        img = image.img_to_array(img)
        # Append the image to the list.
        images.append(img)
    # Return a Numpy array of the images.
    return np.asarray(images) / 255.0
```

Fig.3.8. Sync the dataset to the notebook

3.3.3. Making the GENERATOR and DISCRIMINATOR function

The result of the read_images function will be the input of the generator which will generate 64x64 image and pass it to the discriminator.

These two models are primarily based on the CNN network design.

Beside, a helper function is create to compute cross entropy loss of the training process using tensorflow.

```
# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy()
```

Fig.3.9. Cross entropy

3.3.4. Define the DCGAN model

Methods for building the model, computing the generator and discriminator losses, and training the model using the train_step technique are all included in the DCGAN models. The DCGAN is made up of a generator and a discriminator, and the train_step technique trains the model by alternately training the discriminator and the generator.

Model: "generator"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16384)	2113536
batch_normalization (Batch Normalization)	(None, 16384)	65536
re_lu (ReLU)	(None, 16384)	0
/usr/local/lib/python3.10/dist-packages/keras/initializers/initializers.py:120:		
reshape (Reshape)	(None, 4, 4, 1024)	0
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 512)	13107712
batch_normalization_1 (Batch Normalization)	(None, 8, 8, 512)	2048
re_lu_1 (ReLU)	(None, 8, 8, 512)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 16, 16, 256)	3277056
batch_normalization_2 (Batch Normalization)	(None, 16, 16, 256)	1024
re_lu_2 (ReLU)	(None, 16, 16, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 32, 32, 128)	819328
batch_normalization_3 (Batch Normalization)	(None, 32, 32, 128)	512
re_lu_3 (ReLU)	(None, 32, 32, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 64, 64, 3)	3459
activation (Activation)	(None, 64, 64, 3)	0

=====
Total params: 19,390,211
Trainable params: 19,355,651
Non-trainable params: 34,560

Model: "discriminator"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 128)	3584
leaky_re_lu (LeakyReLU)	(None, 32, 32, 128)	0
conv2d_1 (Conv2D)	(None, 16, 16, 256)	295168
batch_normalization_4 (Batch Normalization)	(None, 16, 16, 256)	1024
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 256)	0
conv2d_2 (Conv2D)	(None, 8, 8, 512)	3277312
batch_normalization_5 (Batch Normalization)	(None, 8, 8, 512)	2048
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 512)	0
conv2d_3 (Conv2D)	(None, 4, 4, 1024)	13108224
batch_normalization_6 (Batch Normalization)	(None, 4, 4, 1024)	4096
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 1024)	0
flatten (Flatten)	(None, 16384)	0
dense_1 (Dense)	(None, 1)	16385
activation_1 (Activation)	(None, 1)	0
=====		
Total params: 16,707,841		
Trainable params: 16,704,257		
Non-trainable params: 3,584		

Fig.3.10. DCGAN network (containing generator and discriminator models)

3.3.5. Training and validating the model

In this step, we create a class that contains functions for compiling, calculating loss, and training the model.

Compile function: This function sets parameters such as optimizer and loss for both the Generator and Discriminator (denoted as 'g' and 'd' respectively).

```
def compile(self, d_optimizer, g_optimizer):
    super(DCGAN, self).compile()
    self.d_optimizer = d_optimizer
    self.g_optimizer = g_optimizer
    self.d_loss_metric = keras.metrics.Mean(name='d_loss')
    self.g_loss_metric = keras.metrics.Mean(name='g_loss')
```

Fig.3.11a. Compile function (inside DCGAN model)

As mentioned, the loss is calculated using `Cross_entropy_loss`. In the Discriminator, the loss is computed as the sum of the fake and real image losses.

```
def generator_loss(self, fake_output):  
    return cross_entropy(tf.ones_like(fake_output), fake_output)  
def discriminator_loss(self, 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)  
    total_loss = real_loss + fake_loss  
    return total_loss
```

Fig.3.11b. Model loss (inside DCGAN model)

Train function for discriminator and generator

- **Train for discriminator**

Step 1: Randomly sample a variable from the latent space to assist in generating a new image.

Step 2: Generate an image using the variables sampled from the latent space.

Step 3: Calculate the loss for the generated (fake) images and the real images as determined by the Discriminator.

```
# Train the discriminator  
with tf.GradientTape() as discriminator_tape:  
    generated_images = self.generator(random_latent_vectors)  
    real_output = self.discriminator(real_images)  
    fake_output = self.discriminator(generated_images)  
    d_loss = self.discriminator_loss(real_output, fake_output)  
    discriminator_grads = discriminator_tape.gradient(d_loss, self.discriminator.trainable_weights)  
    self.d_optimizer.apply_gradients(zip(discriminator_grads, self.discriminator.trainable_weights))
```

Fig.3.11c. Train the discriminator (inside DCGAN model)

Step 4: Update the changes and recompute the loss with the trained weights.

- **Train for generator**

Step 1: Randomly sample a variable from the latent space to assist in generating a new image.

Step 2: Generate an image using the variables sampled from the latent space.

Step 3: Calculate the loss for the generated (fake) images created by the Generator.

Step 4: Update the changes and recompute the loss using the trained weights.

```

# Train the generator
with tf.GradientTape() as generator_tape:
    generated_images = self.generator(random_latent_vectors)
    fake_output = self.discriminator(generated_images)
    g_loss = self.generator_loss(fake_output)
generator_grads = generator_tape.gradient(g_loss, self.generator.trainable_weights)
self.g_optimizer.apply_gradients(zip(generator_grads, self.generator.trainable_weights))

```

Fig.3.11d. Train the generator (inside DCGAN model)

3.3.6. Monitoring the process

In this step, we will create a class that allows the user to monitor the output effectively with customizable parameters such as the number of images and epochs (in this case, it is assumed that every 5 epochs, 10 random images will be monitored).

```

class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim
        self.best_fid = float('inf') # Initialize best_fid with a large value

    def on_epoch_end(self, epoch, logs=None):
        if epoch%5 == 0:
            random_latent_vectors = tf.random.normal(shape=(self.num_img, self.latent_dim))
            generated_images = self.model.generator(random_latent_vectors)
            fig = plt.figure(figsize=(10, 4))

            # Calculate FID using generated images and real images from dataset
            fid = calculate_fid(X_train[:self.num_img], generated_images.numpy())

            print(f'FID at epoch {epoch}: {fid}')
            # Save the model if it achieves the best FID
            if fid < self.best_fid:
                self.best_fid = fid
                self.model.generator.save_weights('/content/best_model.h5')

            for i in range(self.num_img):
                plt.subplot(2, 5, i + 1)
                plt.imshow(generated_images[i,:,:,:]* 0.5 +0.5)
                plt.axis('off')
            plt.show()

```

Fig.3.12. Monitoring the process

Additionally, after a fixed number of epochs, the weights will be compared based on the FID criterion, and the best weights will be saved.

FID will be calculated like below

```
def calculate_fid(real_images, generated_images):
    # Load the Inception-v3 model pre-trained on ImageNet
    model = InceptionV3(include_top=False, pooling='avg')

    # Resize the images to match the minimum input size of InceptionV3
    real_images_resized = np.array([resize(img, (75, 75)) for img in real_images])
    generated_images_resized = np.array([resize(img, (75, 75)) for img in generated_images])

    # Preprocess the images
    real_images_resized = preprocess_input(real_images_resized)
    generated_images_resized = preprocess_input(generated_images_resized)

    # Extract the features from the Inception-v3 model
    real_features = model.predict(real_images_resized)
    generated_features = model.predict(generated_images_resized)

    # Compute the mean and covariance of the real and generated features
    mu_real, sigma_real = np.mean(real_features, axis=0), np.cov(real_features, rowvar=False)
    mu_generated, sigma_generated = np.mean(generated_features, axis=0), np.cov(generated_features, rowvar=False)

    # Compute the squared Euclidean distance between the means
    diff = mu_real - mu_generated
    fid = np.dot(diff, diff) + np.trace(sigma_real + sigma_generated - 2 * linalg.sqrtm(np.dot(sigma_real, sigma_generated)))

    return fid
```

Fig.3.13. Calculating FID

The monitoring results are as follows:




```
Epoch 20/120
220/220 [=====] - 52s 234ms/step - d_loss: 0.9255 - g_loss: 2.0704
Epoch 21/120
220/220 [=====] - ETA: 0s - d_loss: 0.9656 - g_loss: 2.0612WARNING:tensorflow:5 out of the last 9 calls to <
1/1 [=====] - 1s 1s/step
1/1 [=====] - 0s 24ms/step
FID at epoch 20: (26.293406395656206-3.542538392394185e-07j)


220/220 [=====] - 62s 284ms/step - d_loss: 0.9656 - g_loss: 2.0612
Epoch 22/120
220/220 [=====] - 52s 237ms/step - d_loss: 0.8982 - g_loss: 2.1313
Epoch 23/120
220/220 [=====] - 51s 232ms/step - d_loss: 0.9029 - g_loss: 2.0998
Epoch 24/120
220/220 [=====] - 52s 235ms/step - d_loss: 0.8742 - g_loss: 2.1910
Epoch 25/120
220/220 [=====] - 51s 234ms/step - d_loss: 0.8765 - g_loss: 2.2307
Epoch 26/120
220/220 [=====] - ETA: 0s - d_loss: 0.8872 - g_loss: 2.1318WARNING:tensorflow:6 out of the last 11 calls to
1/1 [=====] - 1s 1s/step
1/1 [=====] - 0s 24ms/step
FID at epoch 25: (23.970620469519815-4.3830889135812454e-07j)

```

Fig.3.14. Monitoring example

3.3.7. Declare the parameters for the model and start the training process.

We started by compiling the DCGAN model (using DCGAN class defined above):

```
gan= DCGAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)
gan.compile(
    d_optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
    g_optimizer=Adam(learning_rate=0.0002, beta_1=0.5),)
```

Fig.3.15. Compile the DCGAN model

After that, we save the dataset into X_train:

```
x_train = read_images('/content/pre_img_clean2')
print(x_train.shape)

(7021, 64, 64, 3)
```

Fig.3.16. Read the dataset

Proceed with training (specify parameters such as the number of epochs to train and the number of randomly monitored images).

```
gan_monitor = GANMonitor(num_img=10, latent_dim=latent_dim)

history = gan.fit(x_train, epochs=120, callbacks=[gan_monitor])
```

Fig.3.17. Starting to fit the model

3.3.8. Training result

The first thing to do here is plotting the Generator loss and Discriminator loss to evaluate the training process.

```
# Plot the discriminator and generator loss values over epochs
plt.plot(history.history['d_loss']) # Plot the discriminator loss values
plt.plot(history.history['g_loss']) # Plot the generator loss values

plt.title('Model loss') # Set the title of the plot
plt.ylabel('Loss') # Set the label for the y-axis
plt.xlabel('Epoch') # Set the label for the x-axis

plt.legend(['d_loss', 'g_loss'], loc='upper right') # Add a legend indicating the plotted lines
plt.show() # Display the plot
```

Fig.3.18. Plotting model lost using Matplotlib

The graph results are as follows:

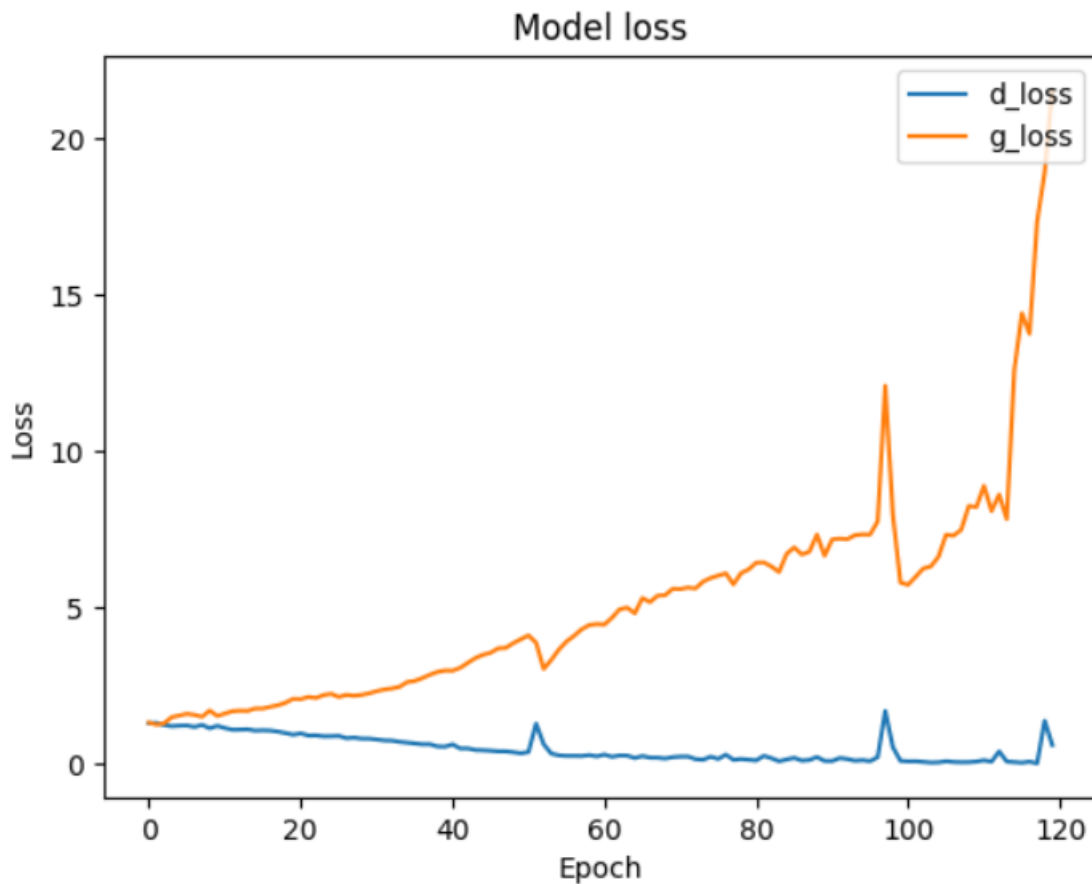


Figure 3.19. Model loss graph

Next, print the outputs using the weights from the best model to verify (in this case, select 16 random images for monitoring).

```
noise = tf.random.normal([16, 128]) # Generate random noise vectors

# generated_images = gan.generator(noise)
gan.generator.load_weights('/content/best_model.h5')
generated_images = gan.generator(noise) # Generate images using the generator model

fig = plt.figure(figsize=(8, 8)) # Create a figure to hold the subplots

for i in range(generated_images.shape[0]): # Iterate over the generated images
    plt.subplot(4, 4, i + 1) # Create a subplot in a 4x4 grid, with index i+1
    plt.imshow((generated_images[i,:,:] * 0.5 + 0.5)) # Display the generated image
    plt.axis("off") # Turn off the axes for the subplot

plt.show() # Display the figure with the generated images
```

The image represents the results:

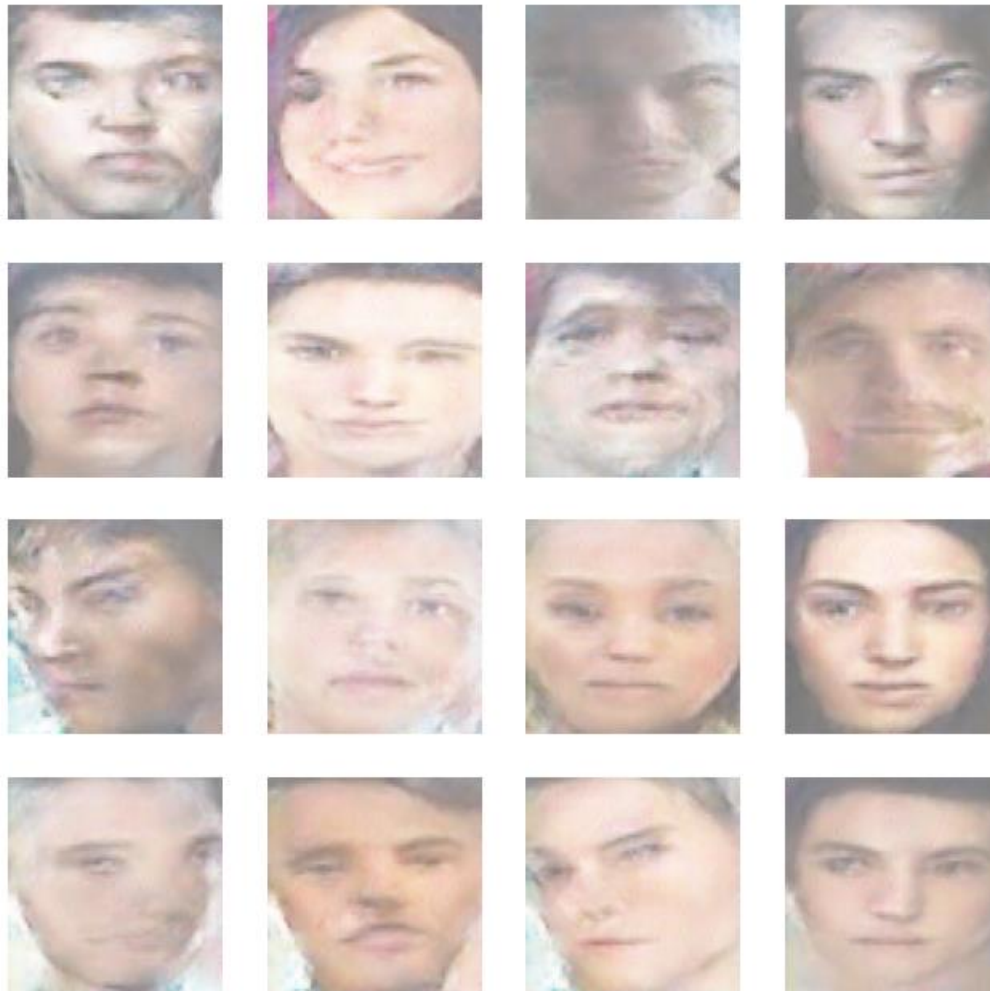


Figure 3.20. Result image of the best model

Finally, we can save the model into .h file using tensorflow library.

CHAPTER 4: CONCLUSION AND FUTURE WORK

4.1. Conclusion

DCGAN's ability to generate new images with high fidelity and its potential for creative applications make it an important tool in the field of generative modeling. Its advancements have paved the way for further research and exploration in image generation, fostering innovation and enhancing the capabilities of artificial intelligence systems.

When using DCGAN to generate an image, some important considerations may include selecting an appropriate image size that aligns with the intended application to achieve the desired results. Additionally, constructing a suitable network architecture is necessary to enhance the effectiveness of the results. During the model training process, there may be undesired variations, so it is essential to have a weight comparison function based on predefined criteria to save the model with the best performance.

4.2. Future work

- We will consider use a better model to train on better hardware
- Try more new application of GAN
- Preprocess the dataset more clearer like the image color, background to improve the output of Generator

REFERENCE

- [1] Brownlee, J. (2019, July 19). *A gentle introduction to generative adversarial networks (Gans)*. MachineLearningMastery.com. <https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/>
- [2] Nguyễn, T. T. (2019, September 17). [GAN series-1] Co ban về GAN trong deep learning. Viblo. <https://viblo.asia/p/gan-series-1-co-ban-ve-gan-trong-deep-learning-bWrZnE4YKxw>
- [3] Working principles of generative adversarial networks (GANs). (2018, September 25). dzone.com. <https://dzone.com/articles/working-principles-of-generative-adversarial-netwo>
- [4] Nguyễn, T. T. (2020, April 2). Deep Convolutional GAN (DCGAN). <https://nttuan8.com/bai-2-deep-convolutional-gan-dcgan>
- [5] Jason Brownlee. (2019, June 14). 18 Impressive Applications of Generative Adversarial Networks (GANs). <https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/>
- [6] DCGAN tutorial — PyTorch tutorials 2.0.1+cu117 documentation. (n.d.). PyTorch. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html