

**A Project Report
on
TEXT TO IMAGE GENERATOR
using
GENERATIVE ADVERSARIAL NETWORKS**

submitted in partial fulfillment of the requirements for the award of the degree of

**BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING**

by

**17WH1A0570 Ms. HARIKA DEVINENI
17WH1A0573 Ms. G. JAHNAVI PRIYA
17WH1A05B0 Ms. S. SAI SWETA**

**under the esteemed guidance of
Mr. R.S. MURALI NATH
Associate Professor, CSE**



**Department of Computer Science and Engineering
BVRIT HYDERABAD College of Engineering for Women
(NBA Accredited – EEE, ECE, CSE and IT)
(Approved by AICTE, New Delhi and Affiliated to JNTU Hyderabad)
Bachupally, Hyderabad – 500090**

May, 2021

DECLARATION

We hereby declare that the work presented in this project entitled “**TEXT TO IMAGE GENERATOR using GENERATIVE ADVERSARIAL NETWORKS**” submitted towards completion of Project Work in IV year of B.Tech. CSE at BVRIT HYDERABAD College of Engineering For Women, Hyderabad is an authentic record of our original work carried out under the guidance of Mr. R.S. MURALI NATH, Associate Professor, Department of CSE.

Sign. with date:
Harika Devineni
(17WH1A0570)

Sign. with date:
G. Jahnvi Priya
(17WH1A0573)

Sign. with date:
S. Sai Sweta
(17WH1A05B0)

BVRIT HYDERABAD College of Engineering for Women
(NBA Accredited – EEE, ECE, CSE and IT)
(Approved by AICTE, New Delhi and Affiliated to JNTU, Hyderabad)
Bachupally, Hyderabad – 500090

Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Project Work report on **“TEXT TO IMAGE GENERATOR using GENERATIVE ADVERSARIAL NETWORKS”** is a bonafide work carried out by Ms. Harika Devineni (17WH1A0570), Ms. G. Jahnavi Priya (17WH1A0573), Ms. S. Sai Sweta (17WH1A05B0) in the partial fulfillment for the award of B.Tech. degree in Computer Science and Engineering, **BVRIT HYDERABAD College of Engineering for Women**, Bachupally, Hyderabad, affiliated to Jawaharlal Nehru Technological University, Hyderabad under my guidance and supervision.

The results embodied in the project work have not been submitted to any other University or Institute for the award of any degree or diploma.

Head of the Department
Dr. Ch.Srinivasulu
Professor and HOD
Department of CSE

Guide
Mr. R.S. Murali Nath
Associate Professor
Department of CSE

External Examiner

Acknowledgements

We would like to express our sincere thanks to **Dr. K V N Sunitha**, Principal, BVRIT HYDERABAD College of Engineering for Women, for providing the working facilities in the college.

Our sincere thanks and gratitude to **Dr. Ch. Srinivasulu**, Professor and HOD, Department of CSE, BVRIT HYDERABAD College of Engineering for Women for all the timely support and valuable suggestions during the period of our project.

We are extremely thankful and indebted to our internal guide, **Mr. R.S. Murali Nath**, Associate Professor, Department of CSE, BVRIT HYDERABAD College of Engineering for Women for his constant guidance, encouragement and moral support throughout the project.

Finally, we would also like to thank our Project Coordinator, all the faculty and staff of CSE Department who helped us directly or indirectly, parents and friends for their cooperation in completing the project work.

Ms. Harika Devineni
(17WH1A0570)

Ms. G. Jahnavi Priya
(17WH1A0573)

Ms. S. Saii Sweta
(17WH1A05B0)

Contents

S.No.	Topic	Page No.
	Abstract	i
	List of Figures	ii
1	Introduction	1
	1.1 Objectives	1
	1.2 Methodology	1
	1.2.1 Dataset	1
	1.2.2 The proposed GAN model	2
	1.3 Organization of Project	5
2	Theoretical Analysis of Proposed Project	6
	2.1 Requirements Gathering	6
	2.2 Technologies Description	6
3	Design	7
	3.1 Introduction	7
	3.2 Architecture Diagram	7
	3.3 UML Diagrams	8
	3.3.1 Use Case Diagram	8
	3.3.2 Activity Diagram	8
4	Implementation	9
	4.1 Coding	9
	4.2 Testing	18
	4.3 Training Screenshots	19
	4.4 Output Screenshots	21
5	Conclusion and Future Scope	22
6	References	22

ABSTRACT

Synthesizing high-quality images from text descriptions is a challenging problem in computer vision and has many practical applications. Samples generated by existing text-to-image approaches can roughly reflect the meaning of the given descriptions, but they fail to contain necessary details and vivid object parts. In this project, we use Stacked Generative Adversarial Networks (StackGAN) to generate photo-realistic images conditioned on text descriptions. The hard problem is decomposed into more manageable sub-problems through a sketch-refinement process. The Stage-I GAN sketches the primitive shape and colors of the object based on the given text description, yielding Stage-I low-resolution images. The Stage-II GAN takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photo-realistic details. It is able to rectify defects in Stage-I results and add compelling details with the refinement process.

LIST OF FIGURES

Figure No.	Figure Name	Page No.
1	Overview of how a GAN network works	3
2	Formula for GAN network	3
3	The architecture of the proposed StackGAN	4
4	Architecture Diagram	7
5	Use Case Diagram	8
6	Activity Diagram	8
7	Training dataset images	19
8	Training Final image caption data	20
9	Model training	20
10	Output of flower with yellow oval shaped petals	21
11	Output of flower with oval shaped pink petals	21

1. INTRODUCTION

One of the most challenging problems in the world of Computer Vision is synthesizing high-quality images from text descriptions. No doubt, this is interesting and useful, but current AI systems are far from this goal. In recent years, powerful neural network architectures like GANs (Generative Adversarial Networks) have been found to generate good results. Generating photo-realistic images from text has tremendous applications, including photo-editing, computer-aided design, etc.

1.1 Objectives

The goal of text to image generator is to enable flexible retrieval between images and text. The fundamental challenge in text to image generator lies in the large visual-semantic discrepancy between images and texts. To generate high-resolution images with photo-realistic details, a simple yet effective system using Stacked Generative Adversarial Networks (StackGAN) is proposed here. The architectures that could help achieve the task of generating images from given text descriptions are explored.

1.2 Methodology

In this system, the text-to-image generative process is divided into two stages. In Stage-I GAN, it sketches the primitive shape and basic colors of the object conditioned on the given text description, and draws the background layout from a random noise vector, yielding a low-resolution image. In Stage-II GAN, it corrects defects in the low-resolution image from Stage-I and completes details of the object by reading the text description again, producing a high-resolution photo-realistic image.

1.2.1 Dataset

Results are presented on the Oxford-102 dataset of flower images having 8,189 images of flowers from 102 different categories. Each image has ten text captions that describe the image of the flower in different ways. Each class consists of between 40 and 258 images. The image descriptions were processed into character level embeddings using the 300D GloVe embeddings.

Introduction

The dataset used has been created with flowers chosen to be commonly occurring in the United Kingdom. The images have large scale, pose and light variations. In addition, there are categories having large variations within the category and several similar categories. The dataset is visualized using isomap with shape and colour features.

Loading GloVe model

GloVe embeddings are used to process image descriptions into character level. **GloVe (Global Vectors for Word Representation)** is an unsupervised learning algorithm for obtaining vector representations for words.

The GloVe model is trained on the non-zero entries of a global word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus. Populating this matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time up-front cost. Subsequent training iterations are much faster because the number of non-zero matrix entries is typically much smaller than the total number of words in the corpus.

The tools provided in this package automate the collection and preparation of co-occurrence statistics for input into the model. The core training code is separated from these preprocessing steps and can be executed independently. We used 300d text embeddings and loaded 400000 words in our system.

Data Pre-Processing

The images were loaded and preprocessed. The processing time for this step was around an hour and half. Because of the longer processing time, the processed files were stored in binary format. This way we can simply reload the processed training data and quickly use it. It is most efficient to only perform this operation once. The dimensions of the image are encoded into the filename of the binary file because it is important to regenerate it if these change. The trained images are stored as numpy.

After images, The captions are loaded and preprocessed. The processing time for this step was around an hour. Because of the longer processing time, the processed files were stored in binary format. The caption embeddings were stored as numpy. A data frame was also created to store and show the captions.

After preprocessing images and captions, both the numpys are loaded and combined. A list of all the preprocessed images was created. Set of images are saved separately for testing. Now the list of the trained data is shuffled. The final dataset is now ready to be fed to the GAN model.

1.2.2 The proposed GAN model

Generative Adversarial Networks

Generative Adversarial Networks (GANs) for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks. Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input

data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models:

The generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

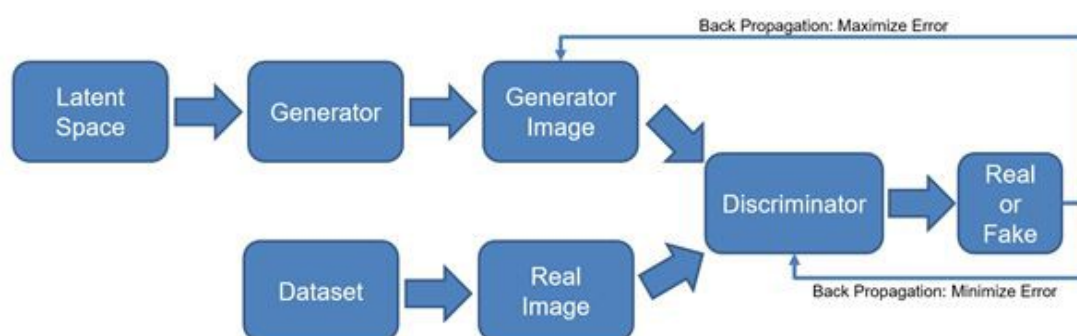


Figure 1: Shows the overview of how a GAN network works

$$\min_G \max_D \left[\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log D(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \sim p_z} \log (1 - D(G(\mathbf{z}))) \right]$$

Figure 2: Equation where \mathbf{z} is a latent "code" that is often sampled from a simple distribution (such as normal distribution). Conditional GAN is an extension of GAN where both generator and discriminator receive additional conditioning variables \mathbf{c} , yielding $G(\mathbf{z}, \mathbf{c})$ and $D(\mathbf{x}, \mathbf{c})$. This formulation allows G to generate images conditioned on variables \mathbf{c} .

Generative Adversarial Text-To-Image Synthesis

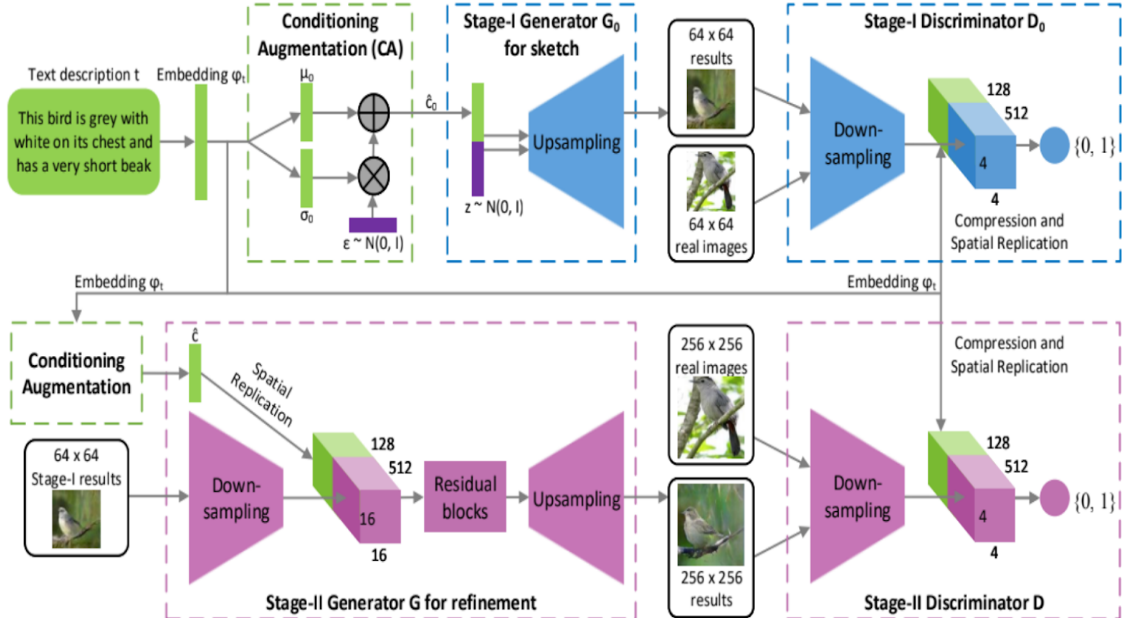


Figure 3: The architecture of the proposed StackGAN.

The Stage-I generator draws a low-resolution image by sketching rough shape and basic colors of the object from the given text and painting the background from a random noise vector. Conditioned on Stage-I results, the Stage-II generator corrects defects and adds compelling details into Stage-I results, yielding a more realistic high-resolution image.

Stage-I GAN Model Architecture

For the generator G_0 , to obtain text conditioning variable \hat{c}_0 , the text embedding ϕ_t is first fed into a fully connected layer to generate μ_0 and σ_0 (σ_0 are the values in the diagonal of Σ_0) for the Gaussian distribution $N(\mu_0(\phi_t), \Sigma_0(\phi_t))$. \hat{c}_0 are then sampled from the Gaussian distribution. Our N_g dimensional conditioning vector \hat{c}_0 is computed by $\hat{c}_0 = \mu_0 + \sigma_0$ (where $+$ is the element-wise multiplication, $\sim N(0, I)$). Then, \hat{c}_0 is concatenated with a N_z dimensional noise vector to generate a $W_0 \times H_0$ image by a series of up-sampling blocks.

For the discriminator D_0 , the text embedding ϕ_t is first compressed to N_d dimensions using a fully-connected layer and then spatially replicated to form a $M_d \times M_d \times N_d$ tensor. Meanwhile, the image is fed through a series of down-sampling blocks until it has $M_d \times M_d$ spatial dimension. Then, the image filter map is concatenated along the channel dimension with the text tensor. The resulting tensor is further fed to a 1×1 convolutional layer to jointly learn features across the image and the text. Finally, a fully connected layer with one node is used to produce the decision score.

Stage-II GAN Model Architecture

Stage-II generator is designed as an encoder-decoder network with residual blocks similar to the previous stage, the text embedding ϕ_t is used to generate the N_g dimensional text conditioning vector \hat{c} , which is spatially replicated to form a $M_g \times M_g \times N_g$ tensor. Meanwhile, the Stage-I result s_0 generated by Stage-I GAN is fed into several down-sampling blocks (i.e., encoder) until it has a spatial size of $M_g \times M_g$. The image features and the text features are concatenated along the channel dimension. The encoded image features coupled with text features are fed into several residual blocks, which are designed to learn multi-modal representations across image and text features. Finally, a series of up-sampling layers (i.e., decoder) are used to generate a $W \times H$ high-resolution image. Such a generator is able to help rectify defects in the input image while adding more details to generate the realistic high-resolution image.

For the discriminator, its structure is similar to that of Stage-I discriminator with only extra down-sampling blocks since the image size is larger in this stage. To explicitly enforce GAN to learn better alignment between the image and the conditioning text, rather than using the vanilla discriminator, we adopt the matching-aware discriminator proposed by Reed et al. for both stages. During training, the discriminator takes real images and their corresponding text descriptions as positive sample pairs, whereas negative sample pairs consist of two groups. The first is real images with mismatched text embeddings, while the second is synthetic images with their corresponding text embeddings.

1.3 Organization of Project

The system developed is taking text descriptions as input and generates realistic images based on the description.

The text description may contain many synonyms, to identify them, the GloVe model was used. For example, Frog may be described as Toad or its scientific name. Model identifies such similar words in the whole dataset, populates the co-occurrence matrix and maps the words to the corresponding vector form.

This vector is now given as input to our GAN model which then performs the task of converting the input vector to image.

2. THEORETICAL ANALYSIS OF THE PROPOSED PROJECT

2.1 Requirements Gathering

2.1.1 Software Requirements

Programming Language : Python 3.6

Dataset : Oxford-102 flower dataset

Packages : GloVe model, Tensor Flow, Numpy, Keras, Matplotlib

Tool : Google Colab

2.1.2 Hardware Requirements

Operating System : Windows 10

Processor : Intel Core i7

CPU Speed : 2.30 GHz

Memory : 2 GB (RAM)

2.2 Technologies Usage

Python 3.6

Python3 was used to code this system as it is convenient and efficient to use and train machine learning models with python.

Tensorflow

Tensorflow Package is imported and used to implement the GAN network using Python and Keras.

Numpy

Numpy is a general-purpose array-processing package. It is used in our system to store the processed images and captions and use them further for training our model.

Keras

Keras was used to implement the Generative Adversarial Network which is in turn nothing but the neural network. Sequential model from keras was used to build the network.

Matplotlib

Matplotlib was used to plot and display the results visually.

3. DESIGN

3.1 Introduction

Text to image Generator system works similar to any other machine learning model. First, the data is preprocessed i.e. The images and captions are normalised, converted to vector form and stored in the form of numpys.

Then the processed data is given as input to the GAN network for training. Stage I and Stage II GANs are built. The stage I GAN takes text description and gives a low resolution image as output which is the outline of the target image. Stage II then takes this blurred image and the text description, matches them to find what is required and adds the finishing details to give out a fully generated image.

Now, the image is passed through many upsampling blocks to generate a high resolution target image.

3.2 Architecture Diagram

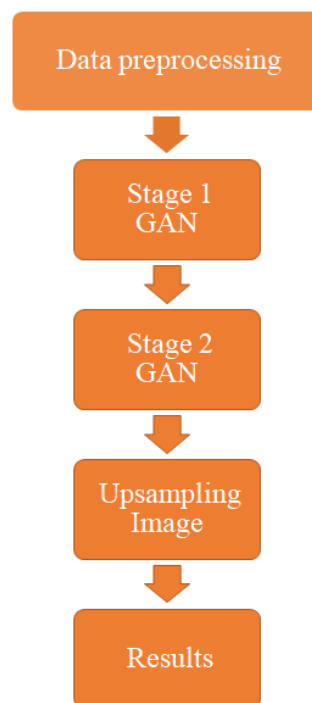


Fig 4: Architecture Diagram

3.3 UML Diagrams

3.3.1 Use Case Diagram

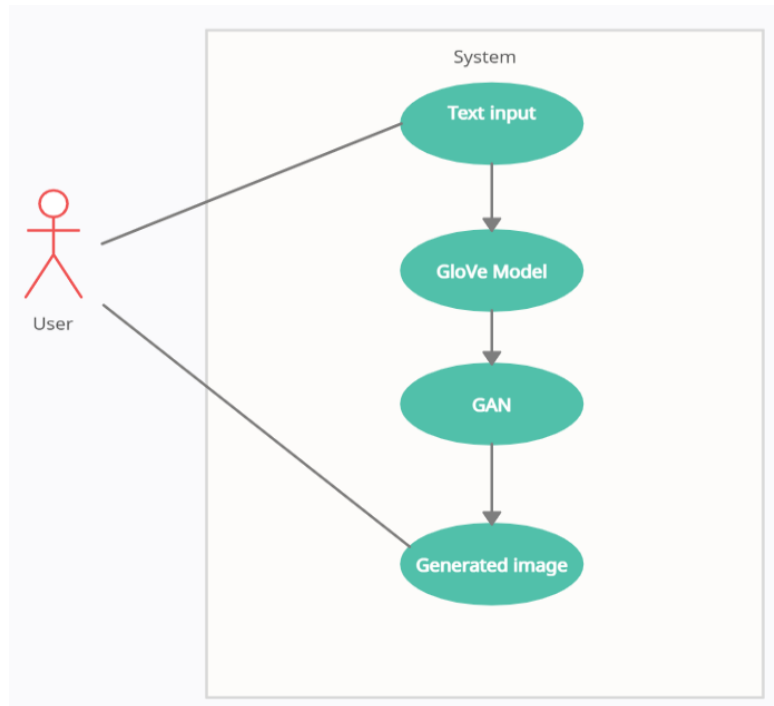


Fig 5: Use Case Diagram

3.3.2 Activity Diagram

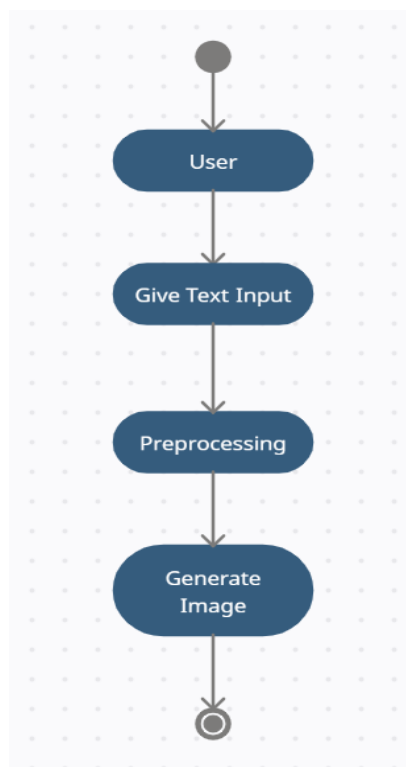


Fig 6: Activity Diagram

4. IMPLEMENTATION

4.1 Coding

Necessary Imports

```
from google.colab import drive
drive.mount('/content/drive')

import glob
import pandas as pd
import urllib.request
import imageio
import os
import numpy as np
from PIL import Image
from tqdm import tqdm
import time
import matplotlib.pyplot as plt
from urllib.request import urlopen

import tensorflow as tf
from tensorflow.keras.layers import Input, Reshape, Dropout, Dense, Concatenate
from tensorflow.keras.layers import Flatten, BatchNormalization
from tensorflow.keras.layers import Activation, ZeroPadding2D
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.layers import UpSampling2D, Conv2D, Conv2DTranspose
from tensorflow.keras.models import Sequential, Model, load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import plot_model
from tensorflow.keras import initializers
from sklearn.metrics import mean_squared_error
```

Loading GloVe model

```
def loadGloveModel(gloveFile):
    print("Loading Glove Model")
    f = open(gloveFile,'r',encoding="utf8")
    model = {}
    for line in f:
        try:
            splitLine = line.split()
            word = splitLine[0]
```



```

        embedding = np.array([float(val) for val in splitLine[1:]])
        model[word] = embedding
    except:
        print(word)
    print("Done.",len(model)," words loaded!")
    return model

glove_embeddings = loadGloveModel("/content/drive/MyDrive/glove.6B.300d.txt")

```

Data processing

```

train_data_path = "/content/drive/MyDrive/flowers"
train_images_path = "/content/drive/MyDrive/flowers/images/jpg"
train_captions_path = "/content/drive/MyDrive/flowers/text_c10"

```

```

def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)

```

```

GENERATE_RES = 2
GENERATE_SQUARE = 32 * GENERATE_RES
IMAGE_CHANNELS = 3
PREVIEW_ROWS = 4
PREVIEW_COLS = 7
PREVIEW_MARGIN = 16
SEED_SIZE = 100
EMBEDDING_SIZE = 300
DATA_PATH = train_images_path
MODEL_PATH = "/content/drive/MyDrive/flowers/w2v_c10"
EPOCHS = 50
BATCH_SIZE = 64
BUFFER_SIZE = 4000

```

```

print(f"Will generate {GENERATE_SQUARE}px square images.")

```

```

training_binary_path = os.path.join("/content/drive/MyDrive/flowers/images/np64",
f'training_data_{GENERATE_SQUARE}_{GENERATE_SQUARE}_')

```

```

start = time.time()
print("Loading training images...")

```

```

training_data = []

```

```

flowers_path = sorted(os.listdir(DATA_PATH))

for filename in range(len(flowers_path)):
    path = os.path.join(DATA_PATH,flowers_path[filename])
    # print(path)
    try:
        image =
Image.open(path).resize((GENERATE_SQUARE,GENERATE_SQUARE),Image.A
NTIALIAS)
        channel = np.asarray(image).shape[2]
        if channel == 3:
            training_data.append(np.asarray(image))
    except KeyboardInterrupt:
        print("Keyboard Interrup by me...")
        break
    except:
        pass
    if len(training_data) == 100:
        training_data = np.reshape(training_data,(-1,GENERATE_SQUARE,
            GENERATE_SQUARE,IMAGE_CHANNELS))
        training_data = training_data.astype(np.float32)
        training_data = training_data / 127.5 - 1.

        print("Saving training image " + str(100000 + filename) + ".npy")
        np.save(training_binary_path + str(100000 + filename) + ".npy",training_data)
        elapsed = time.time()-start
        print (f'Image preprocess time: {hms_string(elapsed)}')
        training_data = []
print("Complete")

text_path = "/content/drive/MyDrive/flowers/text_c10"
text_files = sorted(os.listdir(text_path))
captions = []
caption_embeddings = np.zeros((len(text_files),300),dtype=np.float32)
for filename in range(len(text_files)):
    path = os.path.join(text_path,text_files[filename])
    f = open(path,'r')
    data = f.read()
    data = data.split("\n")
    f.close()
    for d in range(1):
        x = data[d].lower()
        x = x.replace(" ", "")

```

```

captions.append(x)
count = 0
for t in x:
    try:
        caption_embeddings[filename] += glove_embeddings[t]
        count += 1
    except:
        print(t)
        pass
caption_embeddings[filename] /= count

embedding_binary_path =
os.path.join('/content/drive/MyDrive/flowers/images/embedding_npy',f'embedding_d
ata_character.npy')

embedding_binary_path =
os.path.join('/content/drive/MyDrive/flowers/images/embedding_npy',f'embedding_d
ata.npy')
np.save(embedding_binary_path,caption_embeddings)

df_captions = pd.DataFrame([])
df_captions['captions'] = captions[:len(final_images)]

df_captions.head()

captions[:10]

df_captions.to_csv("/content/drive/MyDrive/flowers/text_c10/captions.csv",index=N
one)

embedding_binary_path =
os.path.join('/content/drive/MyDrive/flowers/images/embedding_npy',f'embedding_d
ata.npy')

caption_embeddings = np.load(embedding_binary_path)

caption_embeddings.shape

image_binary_path = "/content/drive/MyDrive/flowers/images/npy64/"
images = os.listdir(image_binary_path)

images[-1]

```

```

final_images = np.load(image_binary_path + images[0])
for i in images[1:]:
    print(i)
    try:
        final_images = np.concatenate([final_images,np.load(image_binary_path + i)],axis
= 0)
    except:
        pass

```

```
final_images.shape
```

```
captions = list(df_captions.captions[:,5])
```

```
len(captions)
```

```

save_images_captions = captions[:28].copy()
save_images_embeddings = np.copy(caption_embeddings[:28])
save_images_npy = np.copy(final_images[:28])

```

```
caption_embeddings = caption_embeddings[:final_images.shape[0]]
```

```
caption_embeddings.shape
```

```
p = np.random.permutation(len(final_images))
```

```

final_images_shuffled = final_images[p]
final_embeddings_shuffled = caption_embeddings[p]

```

```
final_images_shuffled.shape
```

```
final_embeddings_shuffled.shape
```

```

train_dataset = tf.data.Dataset.from_tensor_slices({'images':
final_images,'embeddings':
caption_embeddings}).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

```

Building model

```

def build_generator_func(seed_size,embedding_size, channels):
    input_seed = Input(shape=seed_size)
    input_embed = Input(shape = embedding_size)
    d0 = Dense(128)(input_embed)
    leaky0 = LeakyReLU(alpha=0.2)(d0)

```

```

merge = Concatenate()([input_seed, leaky0])

d1 = Dense(4*4*256,activation="relu")(merge)
reshape = Reshape((4,4,256))(d1)

upSamp1 = UpSampling2D()(reshape)
conv2d1 =
Conv2DTranspose(256,kernel_size=5,padding="same",kernel_initializer=initializers.
RandomNormal(stddev=0.02))(upSamp1)
batchNorm1 = BatchNormalization(momentum=0.8)(conv2d1)
leaky1 = LeakyReLU(alpha=0.2)(batchNorm1)

upSamp2 = UpSampling2D()(leaky1)
conv2d2 =
Conv2DTranspose(256,kernel_size=5,padding="same",kernel_initializer=initializers.
RandomNormal(stddev=0.02))(upSamp2)
batchNorm2 = BatchNormalization(momentum=0.8)(conv2d2)
leaky2 = LeakyReLU(alpha=0.2)(batchNorm2)

upSamp3 = UpSampling2D()(leaky2)
conv2d3 =
Conv2DTranspose(128,kernel_size=4,padding="same",kernel_initializer=initializers.
RandomNormal(stddev=0.02))(upSamp3)
batchNorm3 = BatchNormalization(momentum=0.8)(conv2d3)
leaky3 = LeakyReLU(alpha=0.2)(batchNorm3)

upSamp4 = UpSampling2D(size=(GENERATE_RES,GENERATE_RES))(leaky3)
conv2d4 =
Conv2DTranspose(128,kernel_size=4,padding="same",kernel_initializer=initializers.
RandomNormal(stddev=0.02))(upSamp4)
batchNorm4 = BatchNormalization(momentum=0.8)(conv2d4)
leaky4 = LeakyReLU(alpha=0.2)(batchNorm4)

outputConv =
Conv2DTranspose(channels,kernel_size=3,padding="same",kernel_initializer=initiali
zers.RandomNormal(stddev=0.02))(leaky4)
outputActi = Activation("tanh")(outputConv)

model = Model(inputs=[input_seed,input_embed], outputs=outputActi)
return model

def build_discriminator_func(image_shape, embedding_size):
    input_shape = Input(shape=image_shape)

```

```

input_embed = Input(shape=embedding_size)

conv2d1 =
Conv2D(32,kernel_size=4,strides=2,input_shape=image_shape,padding="same",kernel_initializer=initializers.RandomNormal(stddev=0.02))(input_shape)
leaky1 = LeakyReLU(alpha=0.2)(conv2d1)

drop2 = Dropout(0.25)(leaky1)
conv2d2 = Conv2D(64, kernel_size=4, strides=2,
padding="same",kernel_initializer=initializers.RandomNormal(stddev=0.02))(drop2)
batchNorm2 = BatchNormalization(momentum=0.8)(conv2d2)
leaky2 = LeakyReLU(alpha=0.2)(batchNorm2)

drop3 = Dropout(0.25)(leaky2)
conv2d3 = Conv2D(128, kernel_size=4, strides=2,
padding="same",kernel_initializer=initializers.RandomNormal(stddev=0.02))(drop3)
batchNorm3 = BatchNormalization(momentum=0.8)(conv2d3)
leaky3 = LeakyReLU(alpha=0.2)(batchNorm3)

drop4 = Dropout(0.25)(leaky3)
conv2d4 = Conv2D(256, kernel_size=4, strides=2,
padding="same",kernel_initializer=initializers.RandomNormal(stddev=0.02))(drop4)
batchNorm4 = BatchNormalization(momentum=0.8)(conv2d4)
leaky4 = LeakyReLU(alpha=0.2)(batchNorm4)

dense_embed =
Dense(128,kernel_initializer=initializers.RandomNormal(stddev=0.02))(input_embed
)
leaky_embed = LeakyReLU(alpha=0.2)(dense_embed)
reshape_embed = Reshape((4,4,8))(leaky_embed)
merge_embed = Concatenate()([leaky4, reshape_embed])

drop5 = Dropout(0.25)(merge_embed)
conv2d5 = Conv2D(512,
kernel_size=4,kernel_initializer=initializers.RandomNormal(stddev=0.02))(drop5)
batchNorm5 = BatchNormalization(momentum=0.8)(conv2d5)
leaky5 = LeakyReLU(alpha=0.2)(batchNorm5)

drop6 = Dropout(0.25)(leaky5)
flatten = Flatten()(drop6)
output = Dense(1,activation="sigmoid")(flatten)

model = Model(inputs=[input_shape,input_embed], outputs=output)

```

```

return model

generator = build_generator_func(SEED_SIZE, EMBEDDING_SIZE,
IMAGE_CHANNELS)
generator.load_weights("/content/drive/MyDrive/flowers/model/text_to_image_generator_cub_character.h5")

noise = tf.random.normal([1, 100])
generated_image = generator((noise, caption_embeddings[5].reshape(1, 300)),
training=False)

plt.imshow(generated_image[0, :, :, 0])

image_shape =
(GENERATE_SQUARE, GENERATE_SQUARE, IMAGE_CHANNELS)

discriminator = build_discriminator_func(image_shape, EMBEDDING_SIZE)
discriminator.load_weights((os.path.join(MODEL_PATH, "text_to_image_disc_cub_character.h5")))

decision = discriminator((generated_image, caption_embeddings[5].reshape(1, 300)))
print(decision)

cross_entropy = tf.keras.losses.BinaryCrossentropy()

def discriminator_loss(real_image_real_text, fake_image_real_text,
real_image_fake_text):
    real_loss = cross_entropy(tf.random.uniform(real_image_real_text.shape, 0.8, 1.0),
real_image_real_text)
    fake_loss =
(cross_entropy(tf.random.uniform(fake_image_real_text.shape, 0.0, 0.2),
fake_image_real_text) +
cross_entropy(tf.random.uniform(real_image_fake_text.shape, 0.0, 0.2),
real_image_fake_text))/2

    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(learning_rate=2.0e-4, beta_1 = 0.5)

```

```
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=2.0e-4,beta_1 =
0.5)
```

```
@tf.function
```

Training the model

```
def train_step(images,captions,fake_captions):
    seed = tf.random.normal([BATCH_SIZE, SEED_SIZE],dtype=tf.float32)

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator((seed,captions), training=True)
        real_image_real_text = discriminator((images,captions), training=True)
        real_image_fake_text = discriminator((images,fake_captions), training=True)
        fake_image_real_text = discriminator((generated_images,captions), training=True)

        gen_loss = generator_loss(fake_image_real_text)
        disc_loss = discriminator_loss(real_image_real_text, fake_image_real_text,
real_image_fake_text)

        gradients_of_generator = gen_tape.gradient(\
            gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(\
            disc_loss, discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(
            gradients_of_generator, generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(
            gradients_of_discriminator,
            discriminator.trainable_variables))
    return gen_loss,disc_loss

def train(train_dataset, epochs):
    fixed_seed = np.random.normal(0, 1, (PREVIEW_ROWS * PREVIEW_COLS,
SEED_SIZE))
    fixed_embed = save_images_embeddings

    start = time.time()

    for epoch in range(epochs):
        print("epoch start...")
        epoch_start = time.time()

        gen_loss_list = []
        disc_loss_list = []
```



```

for batch in train_dataset[:1]:
    train_batch = batch['images']
    caption_batch = batch['embeddings']

    fake_caption_batch = np.copy(caption_batch)
    np.random.shuffle(fake_caption_batch)

    t = train_step(train_batch,caption_batch,fake_caption_batch)
    gen_loss_list.append(t[0])
    disc_loss_list.append(t[1])
    print("now")
    g_loss = sum(gen_loss_list) / len(gen_loss_list)
    d_loss = sum(disc_loss_list) / len(disc_loss_list)

    epoch_elapsed = time.time()-epoch_start
    print(f'Epoch {epoch+1}, gen loss={g_loss},disc loss={d_loss},
    {hms_string(epoch_elapsed)}')
    save_images(epoch,fixed_seed,fixed_embed)

generator.save(os.path.join(MODEL_PATH,"text_to_image_generator_cub_character.
h5"))

discriminator.save(os.path.join(MODEL_PATH,"text_to_image_disc_cub_character.h
5"))
    print("model saved")

elapsed = time.time()-start
print ('Training time:', hms_string(elapsed))

train(list(train_dataset.as_numpy_iterator()), 500)

save_images_embeddings.shape

save_images_captions

```

4.2 Testing

```

def test_image(text,num):
    test_embeddings = np.zeros((1,300),dtype=np.float32)

    x = text.lower()

```

```

x = x.replace(" ", "")
count = 0
for t in x:
    try:
        test_embeddings[0] += glove_embeddings[t]
        count += 1
    except:
        print(t)
        pass
test_embeddings[0] /= count
test_embeddings = np.repeat(test_embeddings,[28],axis=0)
noise = tf.random.normal([28, 100])
return save_images(num,noise,test_embeddings)

```

4.3 TRAINING SCREENSHOTS

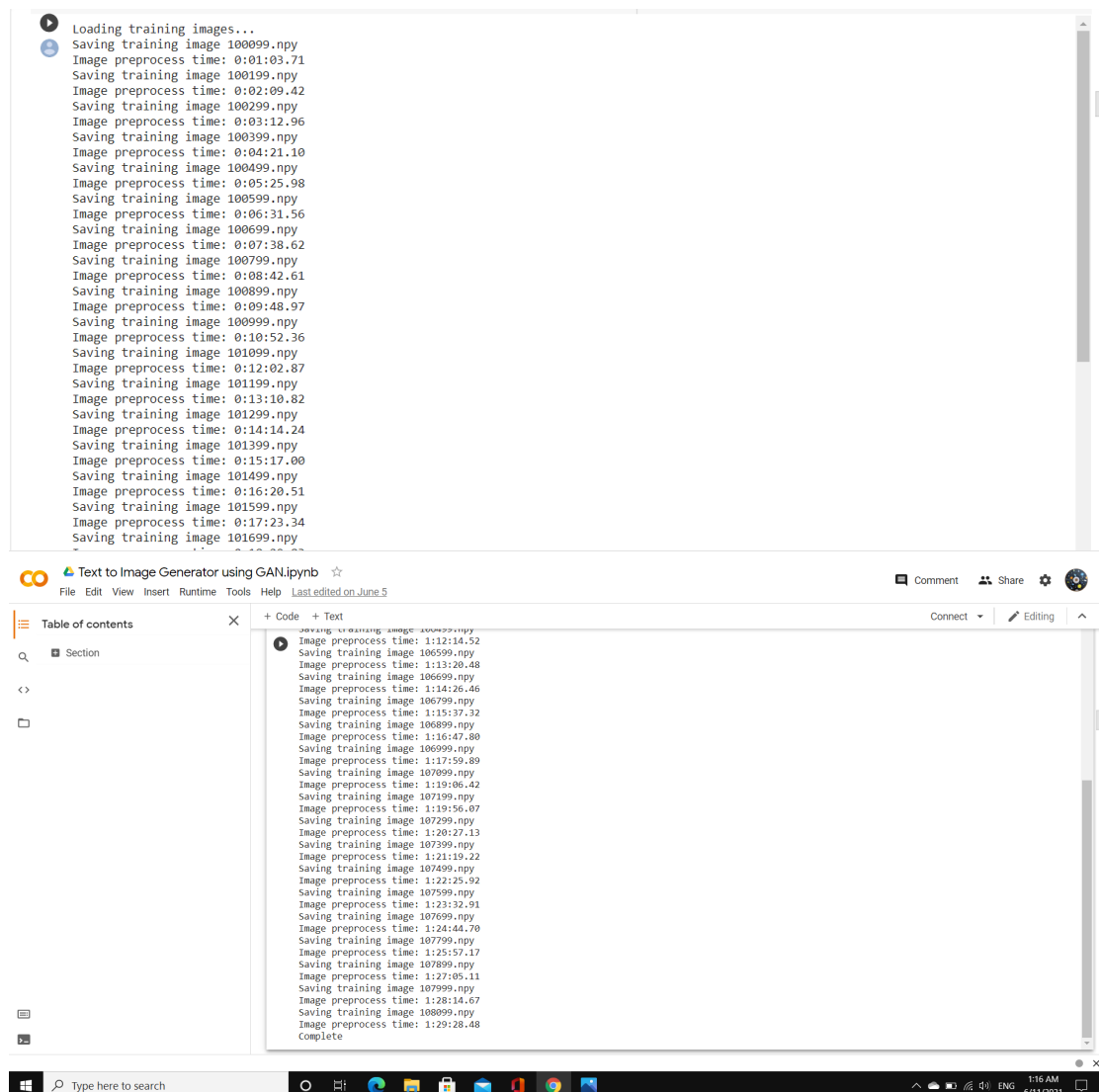
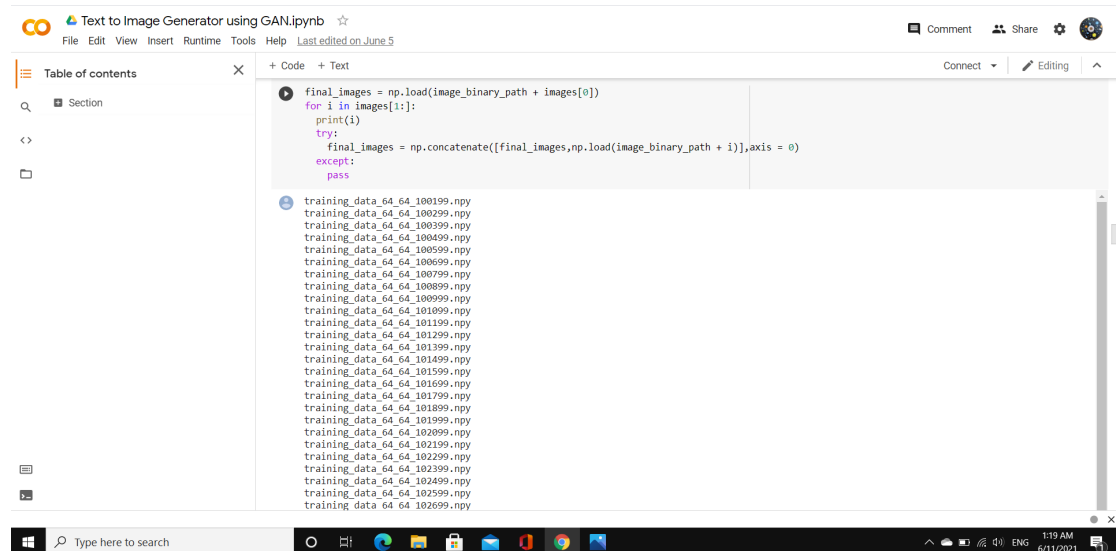


Fig 7: Training dataset images

TEXT TO IMAGE GENERATOR using GENERATIVE ADVERSARIAL NETWORKS



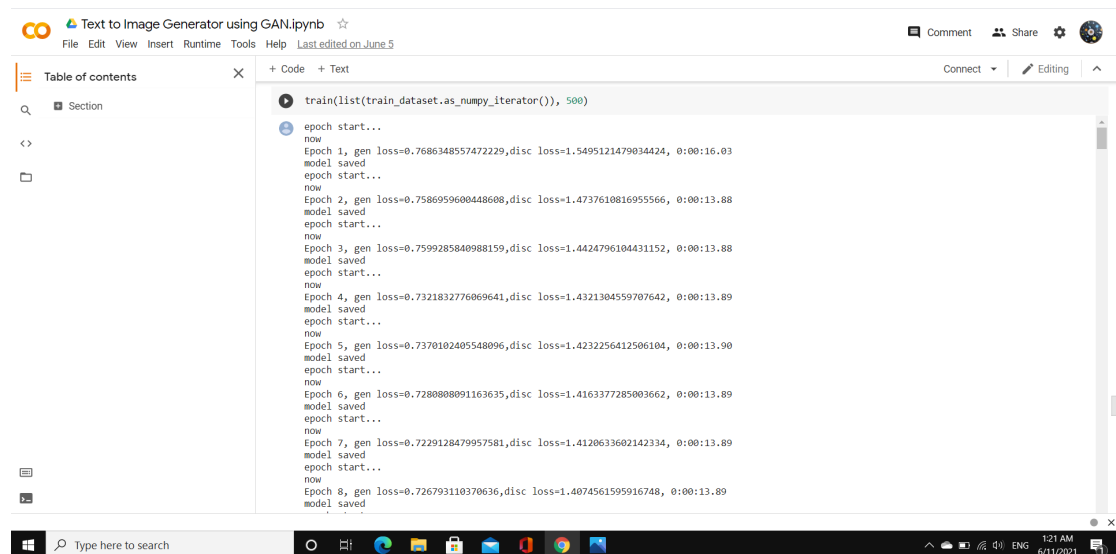
The screenshot shows a Jupyter Notebook titled "Text to Image Generator using GAN.ipynb". The code in the cell is as follows:

```
final_images = np.load(image_binary_path + images[0])
for i in images[1:]:
    print(i)
    try:
        final_images = np.concatenate([final_images, np.load(image_binary_path + i)], axis = 0)
    except:
        pass
```

Below the code, a list of training data files is displayed:

- training_data_64_64_100199.npy
- training_data_64_64_100299.npy
- training_data_64_64_100399.npy
- training_data_64_64_100499.npy
- training_data_64_64_100599.npy
- training_data_64_64_100699.npy
- training_data_64_64_100799.npy
- training_data_64_64_100899.npy
- training_data_64_64_100999.npy
- training_data_64_64_101099.npy
- training_data_64_64_101199.npy
- training_data_64_64_101299.npy
- training_data_64_64_101399.npy
- training_data_64_64_101499.npy
- training_data_64_64_101599.npy
- training_data_64_64_101699.npy
- training_data_64_64_101799.npy
- training_data_64_64_101899.npy
- training_data_64_64_101999.npy
- training_data_64_64_102099.npy
- training_data_64_64_102199.npy
- training_data_64_64_102299.npy
- training_data_64_64_102399.npy
- training_data_64_64_102499.npy
- training_data_64_64_102599.npy
- training_data_64_64_102699.npy

Fig 8: Training Final image caption data



The screenshot shows a Jupyter Notebook titled "Text to Image Generator using GAN.ipynb". The code in the cell is as follows:

```
train(list(train_dataset.as_numpy_iterator()), 500)
```

Below the code, the training progress is displayed, showing the results for each epoch:

- epoch start...
- now
- Epoch 1, gen loss=0.768634857472229, disc loss=1.5495121479034424, 0:00:16.03
- model saved
- epoch start...
- now
- Epoch 2, gen loss=0.7586959600448608, disc loss=1.4737610816955566, 0:00:13.88
- model saved
- epoch start...
- now
- Epoch 3, gen loss=0.7599285840988159, disc loss=1.4424796104431152, 0:00:13.88
- model saved
- epoch start...
- now
- Epoch 4, gen loss=0.7321832776069641, disc loss=1.4321304559707642, 0:00:13.89
- model saved
- epoch start...
- now
- Epoch 5, gen loss=0.7370102405548096, disc loss=1.4232256412506104, 0:00:13.90
- model saved
- epoch start...
- now
- Epoch 6, gen loss=0.7280808091163635, disc loss=1.4163377285003662, 0:00:13.89
- model saved
- epoch start...
- now
- Epoch 7, gen loss=0.7229128479957581, disc loss=1.4120633602142334, 0:00:13.89
- model saved
- epoch start...
- now
- Epoch 8, gen loss=0.726793110370636, disc loss=1.4074561595916748, 0:00:13.89
- model saved

Fig 9: Model training

4.4 OUTPUT SCREENSHOTS

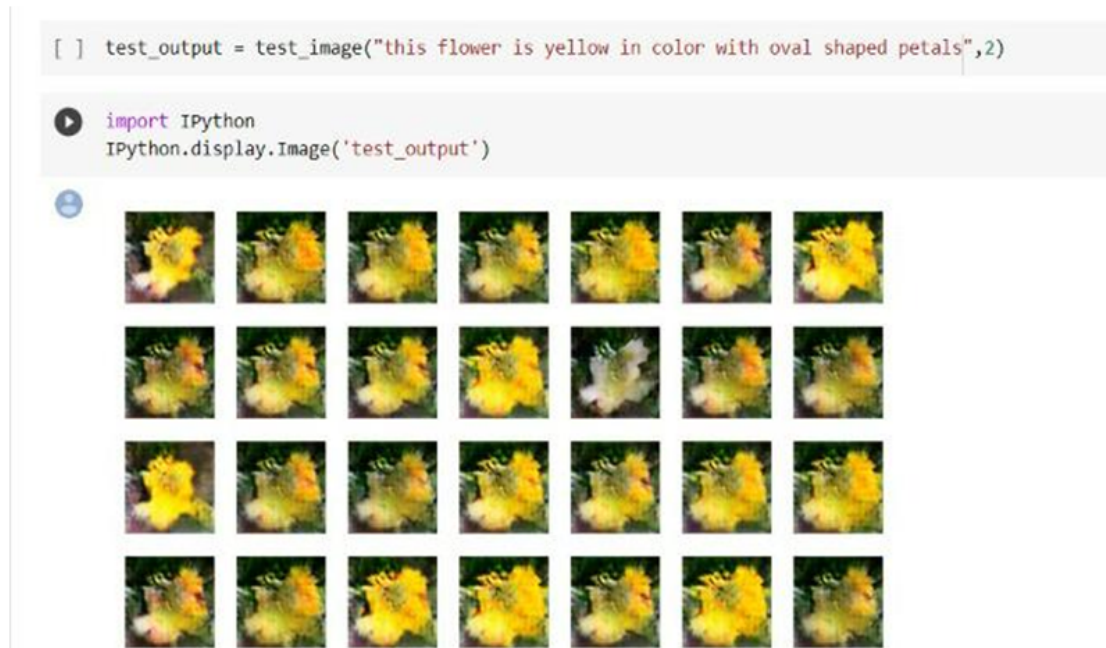


Fig 10: Output of flower with yellow oval shaped petals

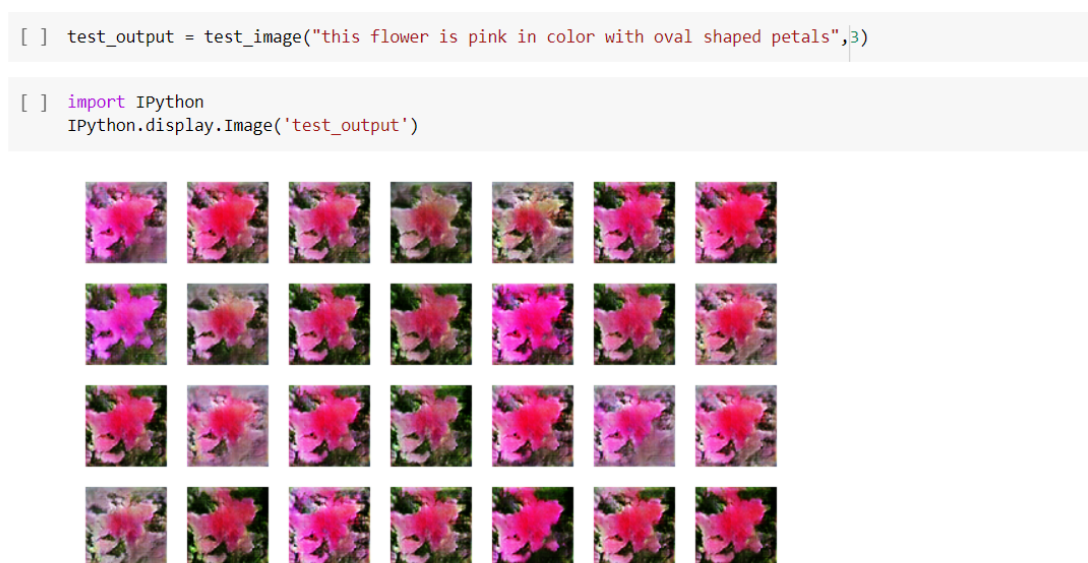


Fig 11: Output of flower with oval shaped pink petals

5. CONCLUSION AND FUTURE SCOPE

Text-to-image synthesis task aims at generating images consistent with input text descriptions and is well developed by the Generative Adversarial Network (GAN). Although GAN based image generation approaches have achieved promising results, synthesizing quality is sometimes unsatisfied due to discursive generation of background and object. In this paper, we propose Stacked Generative Adversarial Networks (StackGAN) with Conditioning Augmentation for synthesizing photo-realistic images. The proposed method decomposes the text-to-image synthesis to a novel sketch-refinement process. Stage-I GAN sketches the object following basic color and shape constraints from given text descriptions. Stage-II GAN corrects the defects in Stage-I results and adds more details, yielding higher resolution images with better image quality. With technology for auto generating images becoming better, it can also be extended to generate various media elements including videos with detailed descriptions.

6. REFERENCES

- [1] “StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks”, TaoXu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, Dimitris Metaxas, Hans Xang date of publication August 5, 2017.
- [2] “TiVGAN: Text to Image to Video Generation With Step-by-Step Evolutionary Generator”, IEEE Access, DOYEON KIM, DONGGYU JOO and JUNMO KIM School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea Corresponding author: Junmo Kim, date of publication August 19, 2020.
- [3] “A Realistic Image Generation of Face From Text Description Using the Fully Trained Generative Adversarial Networks”, MUHAMMAD ZEESHAN KHAN, SAIRA JABEEN, MUHAMMAD USMAN GHANI KHAN, TANZILA SABA, ASIM REHMAT, AMJAD REHMAN and USMAN TARIQ, date of publication August 10, 2020.
- [4] <https://towardsdatascience.com/text-to-image-a3b201b003ae>.