Image Generation from Captions

EEE-443 Neural Network Project

Group Members:

1. Adeem Adil Khatri
2. Taha Khan
3. Talha Naeem
4. Haider Raheem
5. Muhammad Ali Khan

ABSTRACT

Actual image generation and prediction from text captions are essential in the artificial intelligence sector for generation and forecast based on human inputs. Deep Convolutional Generative Adversarial Networks (DCGANs) were an important step forward in this field. Deep Convolutional Generative Adversarial Networks (DC-GAN), in which two models are trained concurrently by an adversarial process, will be thoroughly studied and analyzed. A generator ("the artist") learns to make images that appear genuine, whereas a discriminator ("the art critic") learns to tell the difference between real and fake images. Tensor Layers can be used to create convolutions and de-convolutions for text-image synthesis.

INTRODUCTION

Deep learning techniques require massive data to train effective models for tasks like image recognition and classification. Data augmentation is a deep learning technique that is commonly used in such cases to expand data and prevent over-fitting. This paper looks at how Deep Convolutional Generative Adversarial Networks are used to produce images from given texts. Ian Goodfellow and colleagues proposed the initial Generative Adversarial Networks (GANs) [1]. GANs employ two neural networks: a generator, which uses random noise as input to generate samples (data) as close to the original dataset as possible, and a discriminator, which can distinguish between accurate (actual information) and fake data (generated data).

The DCGAN is a direct extension of the original GAN, with explicit use of convolutional and convolutional-transpose layers in the discriminator and generator. One of the most intriguing aspects of Generative Adversarial Networks is the design of the Generator network. The Generator network can map random noise into images, making it impossible for the discriminator to distinguish between images from the dataset and those from the generator.

In the initial paper, the network uses a 100x1 noise vector, denoted z, and maps it into the 64x64x3 G(Z) output. The network is expanded from 100x1 to 1024x4x4! This layer is referred to as 'project and reshape' [2].

This neural network application appears to be intriguing. Neural nets typically map input into a binary output (1 or 0), a regression output (a real-valued number), or even multiple categorical outputs (like MNIST or CIFAR-10/100). The (N+P — F)/S + 1 equation, typically taught with convolutional layers, reshapes the network with classical convolutional layers.

DATASET

We were provided with 2 Datasets:

1. COCO [2]

   The Dataset consists of photos with 80 classes (such as a person, bicycle, elephant) and a subset of 91 object types (grass, sky, road). We were given the data in two .h5 file formats; one training file and one testing file.

2. Flowers [3]

   We were provided with the Oxford-102 flower dataset as well. With a total of 102 classes, consisting of 20 test classes and 82 training classes. There are 10 text captions for the respective images in the dataset. On this dataset, we ran the DC-GAN architecture with some modifications specific to our problem. We checked and ran the training process

several times with various values of epochs, learning rate, and other hyper-parameter tuning was done to get the best result we could avail with the captions.

Pre Processing of Data

For the Flowers dataset:

1. In order to access and preprocess the data in the flowers dataset. We unzipped the “Flowers.zip”. In it, we had a 102flowers.tgz folder containing all the 8190 images so we extracted data from it too. The captions for each were present in a separate folder “text_c10” and subfolders, “class_*”. We extracted all the captions and sorted them in the order of the images in the 102flowers.tgz folder.
2. Each image had 10 captions associated with it. To train the model, the image dataset consisted of images that were repeated 10 times so that training could be done over each caption. Out of the 8190 images, 819 were separated and compiled as the test set.

GAN(Generative adversarial network)

Goodfellow proposed the generative adversarial network (GAN) as a generative model in 2014 [1]. It comprises two networks: the discriminator D and the generator G. The generator takes a random vector z from a fixed distribution, such as the normal distribution, as input and outputs an image. The discriminator is fed an image and returns a value in (0; 1). During training, the two networks compete, and GAN's goal function is:

$$\min_G \max_D V(D, G) = \min_G \max_D \mathbb{E}_{x \sim pd(x)} [log\ D(x)] + \mathbb{E}_{z \sim pz(z)} [log(1 - D(G(z)))]$$

where:

G = Generator;
D = Discriminator;
x = sample from $\mathbf{P_d\ (x)}$;
D(x) = Discriminator network;
$\mathbf{P_d\ (x)}$ = distribution of real data;
z = sample from **P(z)**;
G(z) = Generator network;
P(z) = distribution of generator;

GAN Architecture

A generative adversarial network (GAN) is composed of two components:

- The **generator's** ability to generate plausible data improves. The generated instances serve as negative training examples for the discriminator.

- The **discriminator** learns to differentiate between true and false data generated by the generator. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces bogus data, which the discriminator quickly detects. As training progresses, the generator gets closer to making output that can fool the discriminator. Finally, if generator training is successful, the discriminator's ability to distinguish between real and fake decreases. It starts mistaking fake data for real, reducing its accuracy. Figure 1 depicts the entire process

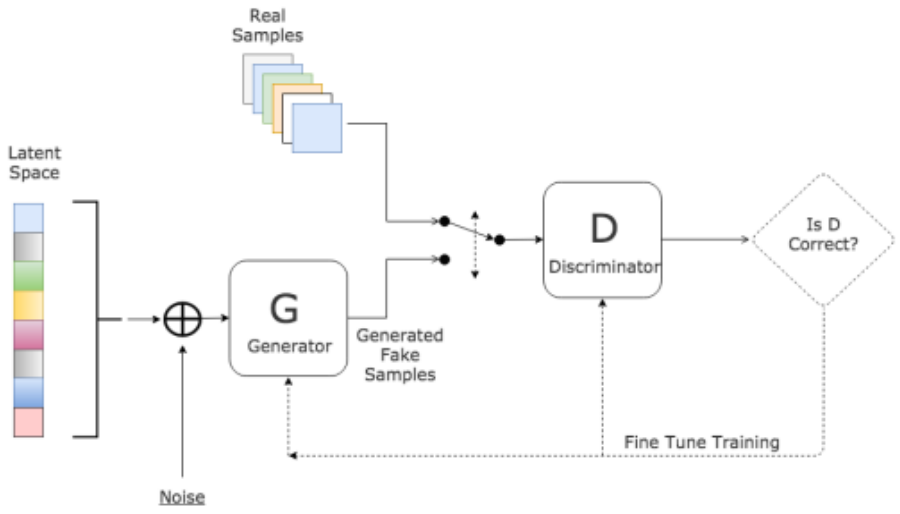


Figure 1. *GAN Architecture*

Both the discriminator and generator are neural networks. The discriminator input is connected to the generator output directly. The generator uses backpropagation to update its weights based on the discriminator's classification.

Working Principle

Generator Goal: Increase the chances that the discriminator will misclassify the generator's output as real.

Discriminator Goal: Optimize for a discriminator goal of 0.5, where the discriminator cannot distinguish between authentic and generated images.

The Minimax Problem (abbreviated MinMax) is a theory that focuses on maximizing a function while incurring the least amount of loss (or vice versa). In the case of GANs, this is represented by the two models training in an adversarial manner. The training step will reduce the error on the generator's training loss while keeping the discriminator as close to 0 as possible (where the discriminator cannot tell the difference between real and fake).

The generator will start training alongside the discriminator in the GAN framework; the discriminator must train for a few epochs before beginning adversarial training because the discriminator must be able to classify images. The structure's final component is the loss function. The loss function determines when the Generator and Discriminator training processes should be terminated.

Generator Steps:

1. To begin, the generator samples from a latent space and establishes a relationship between the latent space and the output. In our case, we also had the word embeddings, which we wanted to map to the image space. Hence, we decided to input the word embeddings into the generator, instead of inputting noise only. The motivation behind this was that the model would train and map the caption embedding space to the image space. Different ways of inputting the embeddings into the generator were tried. They are described in more detail in the Work Done section.

2. Following that, we build a neural network that connects an input (embedding space in our case) to an output (image space).

3. By combining the generator and discriminator into a model, we will train the generator in an adversarial mode (which is how all GANs are trained).

4. The generator can be used to generate images from captions once training is complete.

The model summary of the generator can be seen in the image below.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 131072)            50331648

 batch_normalization (BatchN  (None, 131072)           524288
 ormalization)

 leaky_re_lu (LeakyReLU)     (None, 131072)            0

 reshape (Reshape)           (None, 32, 32, 128)       0

 conv2d_transpose (Conv2DTra  (None, 32, 32, 64)       204800
 nspose)

 batch_normalization_1 (Batc  (None, 32, 32, 64)       256
 hNormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 64)        0

 dropout (Dropout)           (None, 32, 32, 64)        0

 conv2d_transpose_1 (Conv2DT  (None, 64, 64, 32)       51200
 ranspose)

 batch_normalization_2 (Batc  (None, 64, 64, 32)       128
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 64, 64, 32)        0

 conv2d_transpose_2 (Conv2DT  (None, 128, 128, 3)      2400
 ranspose)

=================================================================
Total params: 51,114,720
Trainable params: 50,852,384
Non-trainable params: 262,336
_________________________________________________________________
```

Discriminator Steps:

1. First, it will construct a convolutional neural network to differentiate between real and fake data (binary classification).

2. The images corresponding to the captions that are fed into the generator are fed here, with the aim of training the model such that any given caption can be mapped to its corresponding image upon completion of training.

3. It had to be ensured that the discriminator and generator were actually working adversarially, and that no one was overpowering the other. This was ensured by printing test images at each epoch, to visually observe how effective the training was.

TRAINING METHODS

The idea is to use convolutional and transpose layers instead of dense layers while also introducing constraints like batch normalization, different activation functions for different layers, constraining the optimizer, and designing the upscaling and downscaling layers differently. This results in various convolutional layers capturing multiple essential features, which is beneficial to the overall learning process.

It adheres to a few specific guidelines:

- Strided convolutions (discriminator) and fractional-strided convolutions (generator) are used.
- Both the generator and the discriminator employ Batchnorm.
- Except for the output, all layers in the generator use ReLU activation.
- In the discriminator, LeakyReLU activation is used for all layers.
- The generator output uses a tanh activation function.

Training

The training was carried out over several different parameters which are explained in detail below:

Try 1	**epochs**	**Random noise**	**Learning rate**	**Beta**	**Batch size**	**Image resolution**
	40	0	1*10^-5	0	16	128*128

Table 1. Parameters for training trial 1 on COCO dataset

Try 2	epochs	Random noise	Learning rate	Beta	Batch size	Image resolution
	40	*Standard Gaussian Distribution* with *Mean* = 0 *Variance* = 1	0.002	0.5	128	128*128

Table 2. Parameters for training trial 2 on Flower dataset

Try 3	epochs	Random noise	Learning rate	Beta	Batch size	Image resolution
	40	*Standard Gaussian Distribution* with *Mean* = 0 *Variance* = 1	1*10^-5	0	128	128*128

Table 3. Parameters for training trial 3 on Flower dataset

Try 4	epochs	Random noise	Learning rate	Beta	Batch size	Image resolution
	50	*Standard Gaussian Distribution* with *Mean* = 0 *Variance* = 1	1*10^-5	0	128	128*128

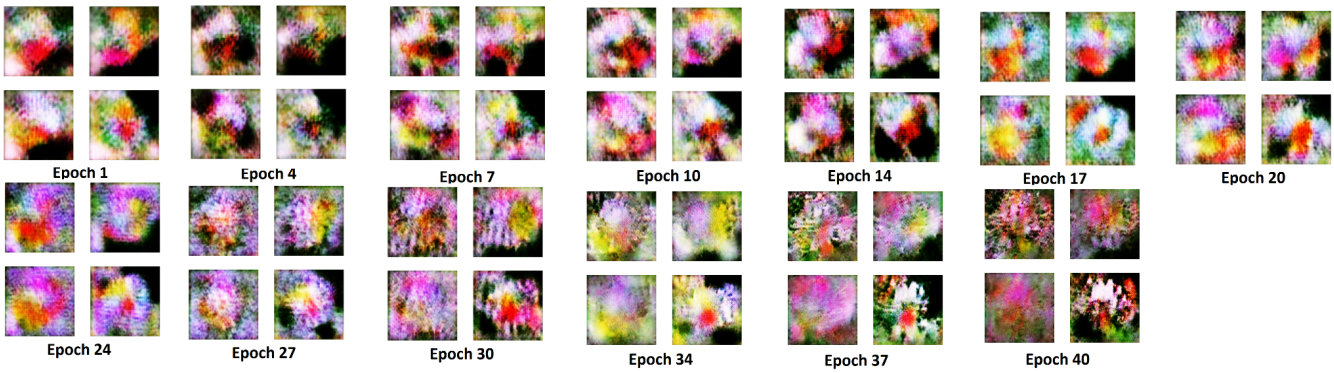
Table 4. Parameters for training trial 4 on Flower dataset

Try 5	epochs	Random noise	Learning rate	Beta	Batch size	Image resolution
	60	*Standard Gaussian Distribution* with *Mean* = 0 *Variance* = 1	0.002	0.5	128	128*128

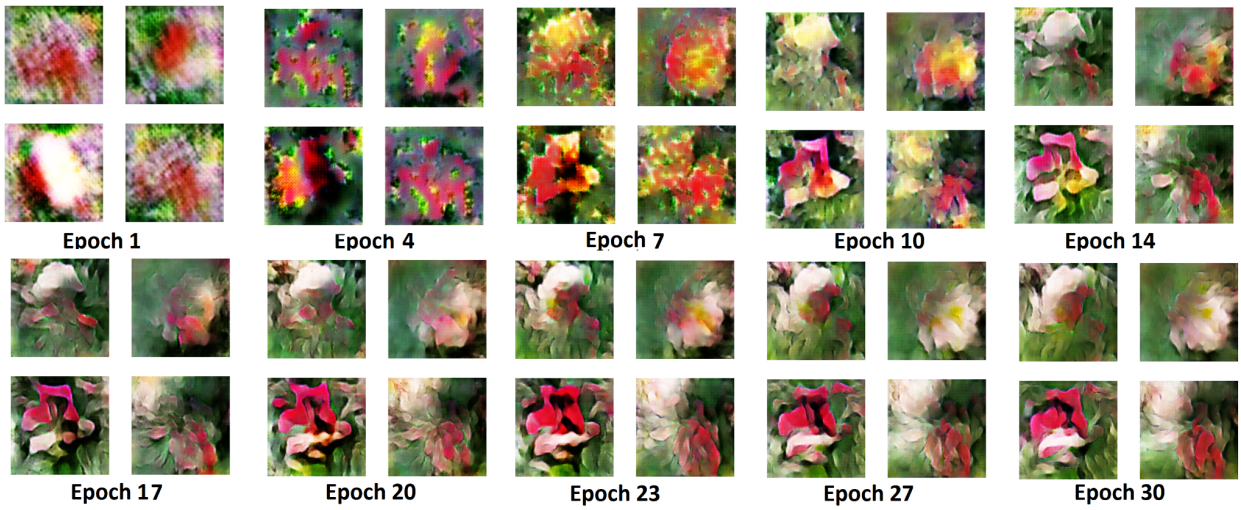
Table 5. Parameters for training trial 4 on Flower dataset

The times of the training varied very little and were around 300 seconds per epoch for every different model that we trained. This is because the number of parameters that were to be trained in the generator and discriminator were kept constant in each case and hence there was no significant effect on training time.

RESULTS & ANALYSIS



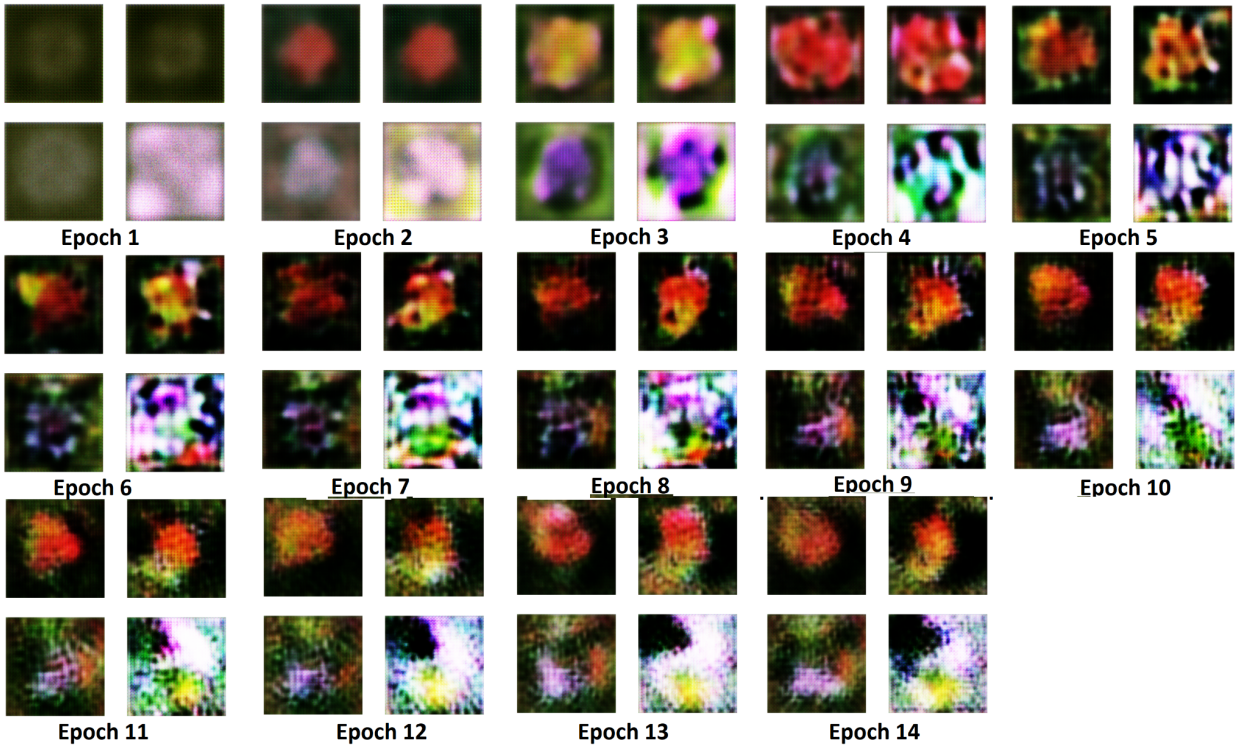
The figure above are the results of the simulation whose parameters are summarized in Table 1. The performance seems to be rather disappointing from start till the end. The model entirely failed to capture the details of the image resolution and quality from the learned captions. As the model does not succeed in producing proper images, it was discarded.



Captions:

1. a white and pink colored flower with a long stigma and a green steam.
2. this white and pink flower has many dense layered petals and a green pedicel.
3. a flower with many pointed pink, white, and red petals with a green center.
4. bright red petals white and green leaf

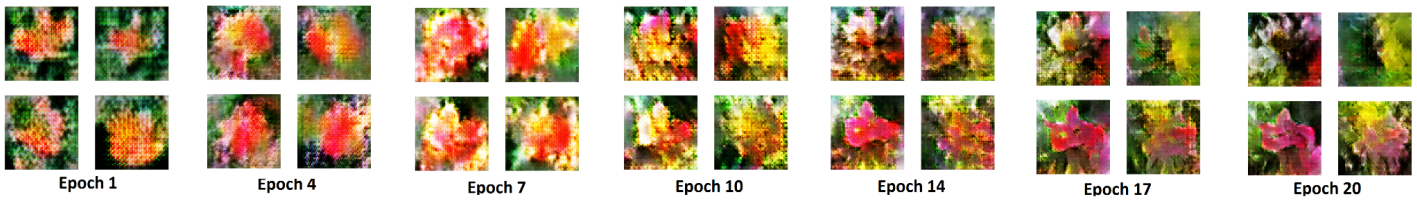
The results that are shown above are validation results of our simulation in Table 2. They were obtained when we introduced *"Gaussian noise at the input only"* (see Appendix) of the DCGAN, and it is evident that after Epoch 4 we start seeing the flower and its resolution and pixel quality improves. The training was stopped after 30 epochs as it can be seen that there is no further learning. It can also be observed that the results are more painting-like, rather than being like real images.



Captions:

1. this flower has a green pedicel and green sepal and white petals as well as dark red filament and anthems
2. many rows of white petals with purple tips are seated on a dark green pedicel & they are surrounding a bright yellow pistil that has bright yellow stamen encasing it.
3. this flower has rounded red petals with white veins, pink and yellow stamen, and a green pedicel.
4. dark red petals green stem

The results that are shown above are validation results of our simulation shown in Table 3. They were obtained when we introduced *"Gaussian noise at the input only"* (see Appendix) of the DCGAN, and it is evident that after Epoch 6, the model starts to overfit and becomes blurry and overpopulates the output. Initially, the model captured the shape and colour of the pixels decently, but from Epoch 7 onwards it started overfitting.

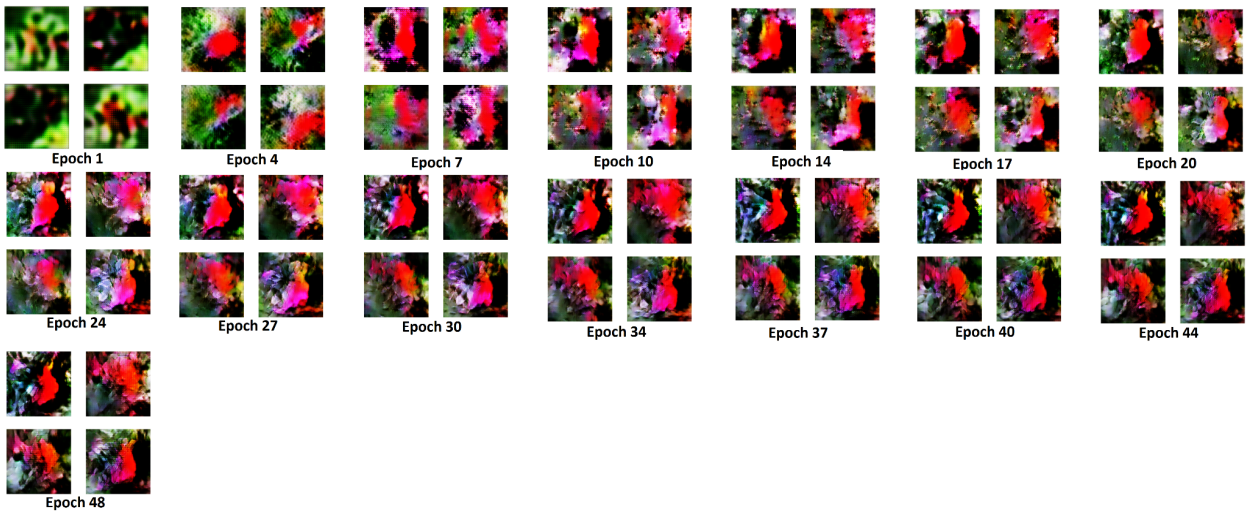


Captions:

1. a flower with pink petals yellow stamen and red stigma and the sepals are green

2. this pale yellow flower with tinges of pink and red throughout has petals that fold inward in a circular, rose pattern, with a few green triangular leaves flaring out behind the main flower.
3. this pink and yellow flower has rounded many layered petals and a green pedicel.
4. this starburst shaped flower has leave-like petals with both red and pink shades, and multiple green pistils and yellow anthers bunched together.

The validation results are shown in the Figure above belong in Table 4. It appears that the model learns slowly but does not overfit and overpopulate the pixels. The quality of the image matches with the caption, but improvement per epoch seems to be very low

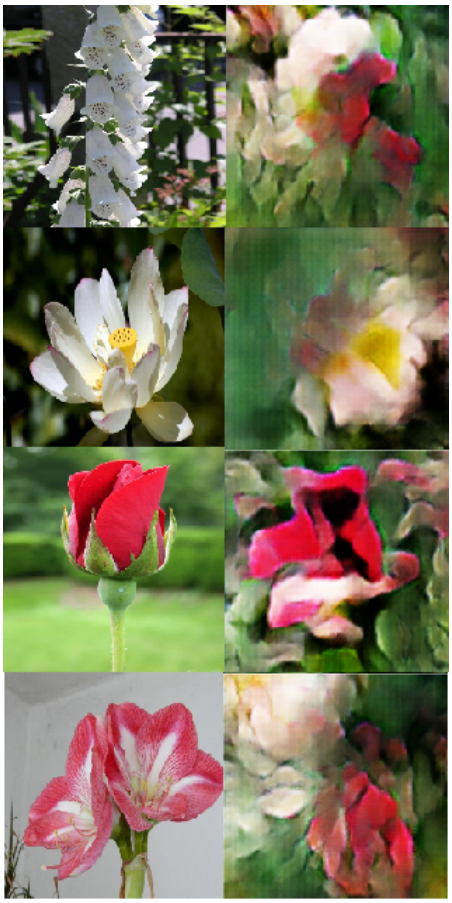


The results above correspond to Table 5 which detects the captions of the similar photos (all red) perfectly. The generated image has improved quality as epochs increase without overfitting and overpopulating the pixels. Even though the images were generated with decent quality, there was a mismatch between captions and generated images. The captions are not shown and we decided to not use this model as a final one.

Performance Metric:

Out of all of these tries, it is quite clear that the model trained for Try 2, is the best performer. To further gauge its performance, the structural similarity index was used. The closer the value is to 1, the more similar the 2 images. The images that were generated after training was completed can be seen below with the actual images in the test set.

Testing



Caption 1: a white and pink colored flower with a long stigma and a green steam.
Caption 2: this white and pink flower has many dense layered petals and a green pedicel.
Caption 3: a flower with many pointed pink, white, and red petals with a green center.
Caption 4: bright red petals white and green leaf

Test Images on left are Actual Image & Test Images on the right are Generated images

Metric	Image 1	Image 2	Image 3	Image 4
SSIM	-0.0040331571	0.00408797216	-0.0012692782	0.017081466786

Looking at the performance metrics, it can be seen that the generated images do not match exactly with the actual images, but these metrics are not accurate in judging the performance of our model. This is because the primary function of our model is to generate images from captions, and not actually match the images to the actual images.

CONCLUSION

Since the DCGAN is one of the earliest models of GANs which was presented in 2014, the model seems to be a good match for generating images of the flower dataset that has a very low variance. Hence, testing such a model for datasets of high variance like COCO is out of the question. It can further be observed that hypertuning the parameters of a model is a very tedious task and requires a lot of resources. Thus was the case and in the end a model that gave outputs corresponding to the input embeddings was developed. Additionally, the images that have high variations in it, e.g. many different shaped objects, were not successfully captured by the model. Reasons for this were observed to be the fact that the DCGAN is a relatively simple architecture which cannot capture high variation regions. The good side of this model is that it takes less time to train and also that it is not as complicated in terms of architecture as other state-of-art models like stackGAN, styleGAN, bigGAN etc. It can therefore be concluded that our model succeeds in carrying out the primary function of the project, which was to generate images from captions. For future works, more complicated models, such as the ones mentioned above can be tried out.

REFERENCES

[1] *Advances in Neural Information Processing Systems 27: 28th Annual Conference on Neural Information Processing Systems 2014 [(NIPS)]; December 8 - 13, 2014, Montreal, Canada; [proceedings of the 2014 Conference]*.

[2] “Common Objects in Context.” *COCO*, https://cocodataset.org/#home.

[3] *102 category Flower Dataset*. Visual Geometry Group - University of Oxford. (n.d.). Retrieved May 27, 2022, from https://www.robots.ox.ac.uk/~vgg/data/flowers/102/

[4] “H5py.” *PyPI*, https://pypi.org/project/h5py/.

APPENDIX

```
pip install torchfile

pip install sentence_transformers

import numpy
import pandas
import shutil
import os
import glob
```

```
import pandas as pd
import torch
import torchfile
import matplotlib.pyplot as plt
import numpy as np
import h5py
from sentence_transformers import SentenceTransformer
import pickle

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

!unzip /content/Flowers.zip

!tar -xvf "/content/Image-to-synthesis-flowers/102flowers.tgz"

!tar -xvf
"/content/Image-to-synthesis-flowers/cvpr2016_flowers.tar.gz" -C
'/content/Image-to-synthesis-flowers'

f = open("/content/Image-to-synthesis-flowers/allclasses.txt", 'r')
classlist = []
for i in f:
  classlist.append(i.strip('\n'))

path_images = "/content/Image-to-synthesis-flowers/text_c10/"
texts = []
for e in classlist:
  texts.append(glob.glob(path_images + e +"/*.txt"))
caption_path =[]
for i in texts:
  for j in i:
    caption_path.append(j)

caption_path[0]

cap_str =[]
for i in caption_path:
  a = open(i, 'r')
  g= (a.read().split('\n'))[:-1]
  #cap_str.append("".join(g))
  cap_str.append(g)

captions_all =[]
for i in cap_str:
```

```
  for j in i:
    captions_all.append(j)

captions_all[6]

captions_all[64]

cap = captions_all

image_names =[]
for i in caption_path:
  image_names.append(((i.split("/")[-1]).split('.')[0]+".jpg"))

model = SentenceTransformer('all-MiniLM-L6-v2')

# caption_embeddings = model.encode(captions_all)

# np.save("caption_embeddings_all", caption_embeddings)

embeddings =
np.load("/content/drive/MyDrive/caption_embeddings_all.npy")

image_paths =[]
img_path = '/content/jpg'
for i in image_names:
  image_paths.append(img_path +"/"+ i)

# def data_loader(i, image_names):
#   img_path = '/content/jpg'
#   x = plt.imread(img_path +'/'+image_names[i])
#   return x

len(image_paths)

image_path_all = []
for i in image_paths:
  for j in range(10):
    image_path_all.append(i)

len(image_path_all)

import shutil
import os

dst_path = '/content/images_final1/'
```

```
os.mkdir(dst_path)
count =0
for i in image_path_all:
  if count < 10:
    shutil.copy(i, (dst_path+ "image_00000"+str(count)+".jpg"))
  elif count >= 10 and count <100:
    shutil.copy(i, (dst_path+ "image_0000"+str(count)+".jpg"))
  elif count >=100 and count <1000:
    shutil.copy(i, (dst_path+ "image_000"+str(count)+".jpg"))
  elif count >=1000 and count < 10000:
    shutil.copy(i, (dst_path+ "image_00"+str(count)+".jpg"))
  elif count >=10000:
    shutil.copy(i, (dst_path+ "image_0"+str(count)+".jpg"))

  count = count +1
    # img_path_all.append(i)

#!zip -r '/content/images_final1.zip' '/content/images_final1'

f = glob.glob('/content/images_final1'+ "/*.jpg")

len(f)

f[0]

f = os.listdir("/content/images_final1")

len(f)

len(image_path_all)

import tensorflow as tf
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
tf.__version__
```

```
train_images =
tf.keras.utils.image_dataset_from_directory("/content/images_final1/"
,labels=None,label_mode=None,class_names=None,color_mode='rgb',batch_
size=128,image_size=(128,
128),shuffle=False,seed=None,validation_split=None,subset=None,interp
olation='bilinear',follow_links=False,crop_to_aspect_ratio=False)

def process(image):
    image = tf.cast((image-127.5)/127.5,tf.float32)
    return image
train_images = train_images.map(process)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images in train_images.take(1):
  for i in range(0,9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.axis("off")

embeddings_tf = tf.convert_to_tensor(embeddings)

embeddings.shape

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(32*32*128, use_bias=False,
input_shape=(384,)))
    model.add(layers.GaussianNoise(1))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((32, 32, 128)))
    assert model.output_shape == (None, 32, 32, 128)  # Note: None is
the batch size

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(1, 1),
padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.5))
```

```
    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
    assert model.output_shape == (None, 64, 64, 32)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2),
padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 128, 128, 3)

    return model

generator = make_generator_model()
import statistics

noise = tf.random.normal([1, 384])
generated_image = generator(noise, training=False)
xyz=tf.reshape(generated_image,(128,128,3))
xyz = tf.cast((xyz*127.5+127.5),tf.int32)
plt.imshow((xyz))

generator.summary()

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2),
padding='same',
                                     input_shape=[128, 128, 3]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.4))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2),
padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.4))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(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

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

generator_optimizer = tf.keras.optimizers.Adam(0.0002, beta_1=0.2)
discriminator_optimizer = tf.keras.optimizers.Adam(0.0002,
beta_1=0.2)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint =
tf.train.Checkpoint(generator_optimizer=generator_optimizer,

discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images,embed):
    #noise = tf.random.normal([256, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as
disc_tape:
      generated_images = generator(embed, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    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_discriminato
r, discriminator.trainable_variables))

def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  predictions = model(test_input, training=False)
  predictions = predictions*127.5+127.5
  predictions = tf.cast(predictions,tf.int32)
  fig = plt.figure(figsize=(4, 4),dpi=200)

  for i in range(predictions.shape[0]):
      plt.subplot(2, 2, i+1)
      plt.imshow(predictions[i, :, :, :])
      plt.axis('off')

  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()

dataset,b, epochs,df = train_images, embeddings_tf, 40, cap

import random
zxcv = random.sample(range(0,81890),4)
for epoch in range(epochs):
  start = time.time()
  count = 0
  for image_batch in dataset:
    for i in range(count,count+128):
      emb = b[i]
      emb = tf.reshape(emb,(1,384))
      emb = emb + 0.001104205
      emb = emb / (0.05101909)
      #noise = tf.random.normal((1,384), stddev=0.9)
      emb = emb + noise
      count += 1
      if count % 128 == 1:
        embed = emb
      else:
        embed = tf.concat([embed,emb],0)
      if count == 81890:
        break
```

```
    train_step(image_batch,embed)
    if count == 81890:
        break

  # Produce images for the GIF as you go
  display.clear_output(wait=True)
  cnt = 0
  for l in zxcv:
    test = b[l]
    test = tf.reshape(test,(1,384))
    test = test + 0.001104205
    test = test  / (0.05101909)
    cnt += 1
    if cnt == 1:
      test_set = test
    else:
      test_set = tf.concat([test_set,test],0)
    capt =  df[l]
    print(capt)
  generate_and_save_images(generator,
                           epoch + 1,
                           test_set)

  # Save the model every 15 epochs
  if (epoch + 1) % 5 == 0:
    checkpoint.save(file_prefix = checkpoint_prefix)

  print ('Time for epoch {} is {} sec'.format(epoch + 1,
time.time()-start))
# Generate after the final epoch
display.clear_output(wait=True)
generate_and_save_images(generator,
                         epochs,
                         test_set)

len(cap)

def get_index(the_list, substring):
    temp = []
    for i, s in enumerate(the_list):
          if substring[0] in s and substring[1] in s and substring[2]
in s and substring[3] in s:
              temp.append(i)
    return temp
```

```
res = get_index(cap, ['white', 'red', 'bright', 'red petals'])

len(res)

res

img_no = 69560
cap[img_no]

path1 = '/content/images_final1/image_0' + str(img_no) + '.jpg'
im1 = Image.open(path1)
im1.show('image',im1)

im1

resul_img = open('/content/images_final1/image_075321.jpg')

from PIL import Image

im = Image.open('/content/images_final1/image_075321.jpg')
im.show('image',im)

im

cap

cap

img1 = open('/content/drive/MyDrive/Copy of Test3_image1.png')
img12 = open('/content/drive/MyDrive/Copy of Test3_image2.png')
img13 = open('/content/drive/MyDrive/Copy of Test3_image3.png')
img14 = open('/content/drive/MyDrive/Copy of Test3_image4.png')



from skimage.metrics import structural_similarity as compare_ssim
#from skimage.measure import compare_ssim
import argparse
import imutils
import cv2

# 2. Construct the argument parse and parse the arguments
#ap = argparse.ArgumentParser()
#ap.add_argument("-f", "--first", required=True, help="Directory of
the image that will be compared")
```

```
#ap.add_argument("-s", "--second", required=True, help="Directory of
the image that will be used to compare")
#args = vars(ap.parse_args())

# 3. Load the two input images
#imageA = cv2.imread(args["first"])
#imageB = cv2.imread(args["second"])

# 4. Convert the images to grayscale
grayA = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
grayB = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# 5. Compute the Structural Similarity Index (SSIM) between the two
#    images, ensuring that the difference image is returned
(score, diff) = compare_ssim(grayA, grayB, full=True)
diff = (diff * 255).astype("uint8")

# 6. You can print only the score if you want
print("SSIM: {}".format(score))

import torch
from PIL import Image
import torchvision.transforms as transforms

# Read the image
img1 = Image.open('im7.jpg')
img2 = Image.open('im6 (1).jpg')
# Define a transform to convert the image to tensor
transform = transforms.ToTensor()

# Convert the image to PyTorch tensor
tensor1 = transform(img1)
tensor2 = transform(img2)

import lpips
loss_fn_alex = lpips.LPIPS(net='alex') # best forward scores
loss_fn_vgg = lpips.LPIPS(net='vgg') # closer to "traditional"
perceptual loss, when used for optimization
d = loss_fn_alex(tensor1, tensor2)
print(d)
```