**BILKENT UNIVERSITY**

**DEPARTMENT OF ELECTRICAL AND ELECTRONICS**

**ENGINEERING**


# EEE 443 Neural Networks -Final Project Report

**11.06.2023**

**Oğuz Can Duran - 21803175**

**ABSTRACT**

In this project text-to-image synthesis is investigated using various Generative Adversarial Networks (GAN) architectures which are DCGAN and DALL-E. The preprocessing of the data for each model is done separately. Since the training process takes a lot of time for DCGAN it is trained for fewer epoch than DALL-E. Since DALL-E is a much more complex model, it generated better images. Additionally, DALL-E is trained with 256x256 images while DCGAN is trained with a resolution of 64x64 due to GPU constraints. At the end it is found that DALL-E is better because of its better performance with short training duration.

## 1.INTRODUCTION

Neural networks are powerful Artificial Intelligence (AI) models that can be used for various tasks, such as image and video synthesis, image captioning and retrieval, object detection, and text-to-image synthesis.

Text-to-image synthesis is a process to create an image from a text description. This task can be done using Generative Adversarial Networks (GANs). GANs are a type of neural network which consists of two different parts: a generator and a discriminator. While the generator creates images, the discriminator tries to distinguish between real and generated (or fake) images.

In this project, I have used different GAN architectures, also evaluated, and compared the models using both qualitative and quantitative methods.

The results showed that DALL-E was better GAN architecture for text-to-image synthesis than DCGAN architecture. This is because it was able to create better images from a variety of text descriptions and its training duration was much shorter.

## 2.METHODS

### 2.1 DATASET

The dataset used for this project was designed for text-to-image synthesis tasks. The train dataset consists of 82,783 images from Flickr, with their corresponding image URLs and 400,135 captions. The arrays given for this task is shown below.

- train cap: Array of captions for the training images. Each caption consists of 17 indices.
- train imid: Array contains the indices of the training images. It is important for associating images with their respective captions, as an image may have multiple captions.
- word code: Serves as the dictionary for converting words into vocabulary indices and indices to words.
- train url: This contains the URLs of the provided images. These URLs were used to download and save the images.

The images were saved according to their indices. However, in the downloading process there was a problem. Some of the images were missing from the provided website. Therefore, while there are 82,783 URLs in the beginning, I could only download 70,312 images for the train dataset.

Similar to train dataset, we were also provided test dataset. This dataset consists of 40,504 images from Flickr, with their corresponding image URLs. Similarly this dataset also includes test_cap, test_imid, and test_url. The images are also downloaded however, some of them were missing again. Therefore, only 34,376 images were downloaded. However, those data were not used in the training process.

Since I can only download an amount of data, I had to update, cap and imid datasets. Therefore, they are also updated according to indices of downloaded images.

The word dictionary comprises 1,004 words. There are special words for the beginning and end of the captions which are (x START and x END), also there are special words for unknown and null words (x UNK and x NULL). These indicators are taken into account in the training processes.

For the task, different preprocessing setups are used. Firstly, a batch size of 64 was used for DCGAN architecture. Also the sizes of images were 64x64 since the higher resolutions were exceeding the GPU memory of Colab and Kaggle. Secondly, a batch size of 32 was used for DALLE architecture. Also the sizes of images were 256x256.

The splitting of dataset and normalization were same for both of the models. As mentioned in the task 15% of the data was separated for validation and 85% for training dataset. Additionally, the for normalization I have used imagenet Mean and Standard deviation which are mean=(0.485,0.456,0.406) and std=(0.229,0.224,0.225).

### 2.2 TASK

Text-to-image generation is a task which combines natural language processing (NLP) and computer vision techniques. It involves taking a text description of an image and generating an image that corresponds to the description. The goal of text to image is to create an image that accurately depicts the details mentioned below.

Text-to-image generation is the process of creating an image from a text description. There are many different approaches to text-to-image generation, but the common method is to use deep neural networks such as Generative Adverserial Netrowk (GAN) or Variational Autoencoder (VAE). During the training process, a text encoder converts takes a text description as input and outputs a latent code. The latent code is a lower-dimensional representation of the text description, which makes it easier for the decoder to generate an image.Then the decoder transforms this representation back into an image. GANs) or VAEs are often used as decoders, as they have been trained to produce visually realistic images which align with the given captions.

When the model is trained, it can generate new images according to text descriptions that the model has not seen before. Although these generated images may not be same with the training photos, they should be similar and consistent with the provided captions.

Briefly, in this project we were asked to create a text-to-image model using the provided datasets. In order to perform this task, I have decided to use DCGAN that uses GAN and DALLE that uses VQGANVAE.

### 2.3 MODELS

#### 2.3.1 GAN

Generative adversarial networks (GANs) are one of the deep learning models that can be used to generate fake realistic data. GANs work by training two neural networks against each other such that both

of those try to reach minimum loss. The first network, called the generator network, is responsible to create new images, while the second network, called the discriminator network, which is responsible for identify the real and fake data.

The generator is trained to create data which is as realistic as possible and send them to discriminator, while the discriminator is trained to identify fake data. Over time, the generator becomes much better in creating realistic fake images, while the discriminator becomes better at distinguishing real and fake data. This process continues until the generator is able to create data that is indistinguishable in discriminator from real data.[1]

Briefly, GANs are being used to generate various fake data, such as images, audio, text and so on. In the context of text-to-image synthesis, GANs have been shown to be particularly well-suited for creating realistic images from text descriptions.

#### 2.3.1.1 DCGAN

Convolutional neural networks (CNNs) are a type of artificial neural network that have been shown to be very effective for computer vision tasks. CNNs are specifically designed to process data with a kernel structure. This makes them good for learning of representations of image data. Additionally, CNNs have the ability to learn features in different scales. This property enables CNN to identify objects in frames whatever their orientations or sizes are. This process are utilized by filters. In this way the model can detect features in various scales.

The paper "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" [2]demonstrates that GANs can be used to train CNNs in an unsupervised manner. Therefore, the use of GANs with a base CNN architecture is a promising approach for the text-to-image task.

The generator network creates a 4 dimensional data using a gaussian random noise. This data is though as image and a series of strided convolutional transpose are applied then each are paired with 2D batch normalization then ReLU activation function. Then the output of the generator is normalized within the range of [-1, 1] using a hyperbolic tangent (tanh) function. This is the desired input data range for the generator. It is important to note that the DCGAN paper introduced the use of batch normalization functions after the convolutional transpose layers. These batch normalization layers help to maintain gradient flow, explosion or vanishing during the training process. The DCGAN paper includes an illustration of the generator architecture, which is shown below[3].

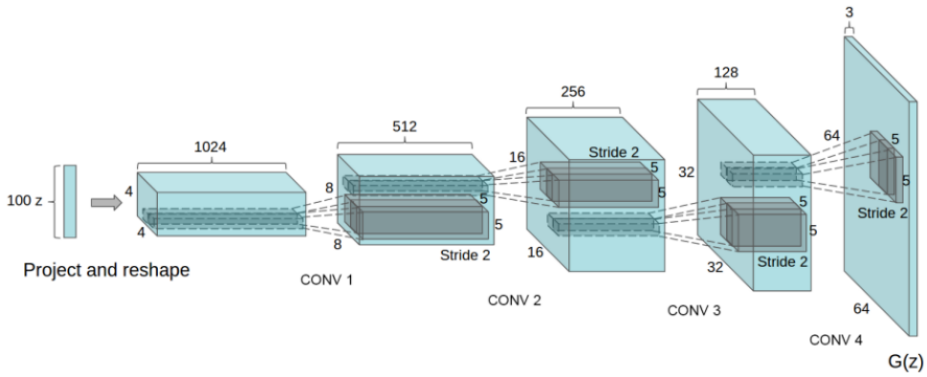


*Figure 1: Illustration of the generator architecture[3]*

In order to generate images with the model, sentences are utilized as prompts. However, directly inputting sentences into the model is not possible, so text embeddings are employed to convert the input sentences. In order to achieve this probem, a pre-trained model which is Bidirectional Encoder Representations from Transformers (BERT) is used. The text embeddings, obtained from BERT, are then combined with the images and sent into the GAN architecture [4].

The Generator model firstly projects the embedded captions from 512 to 128 dimensions. It then combine the noise vector and embeddings to form the latent vector. The latent vector is further processed through four layers of strided convolution transpose, where the hidden layers use ReLU activation, and the output activation function is Tanh as mentioned above.

The Discriminator Network is a CNN-based classifier. It first obtains a tensor from real data then applies four layers of strided convolutions to data. The captions are also projected from 512 to 128 dimensions and batch normalization is applied. Then it passes through LeakyReLU with a slope of 0.2. Obtained two data are combined and followed by a strided convolution layer. Finally, the output is obtained passing through Sigmoid activation function.

### 2.3.2 DALL-E

DALL-E is a OpenAI developed powerful deep learning model for generating images from text descriptions. This model can create realistic and high-resolution images. It is built on a transformer architecture, which consists of two stages: a discrete VAE (Variational Autoencoder) and an autoregressive transformer.

The discrete VAE is responsible for generating a latent representation of the data, while the autoregressive transformer is responsible for generating the actual pixels of the image. The discrete VAE ensures that the generated images are realistic and consistent with the text description, while the autoregressive transformer ensures that the images are high-resolution and detailed. Therefore both of them are required in DALL-E architecture.

Variational Autoencoders (VAEs) are machine learning models that can be used to learn the latent representation of data. VAEs consist of two parts: an encoder and a decoder. The encoder is responsible for converting the input data to a lower-dimensional latent representation, while the decoder reconstruct the original data from the latent representation.

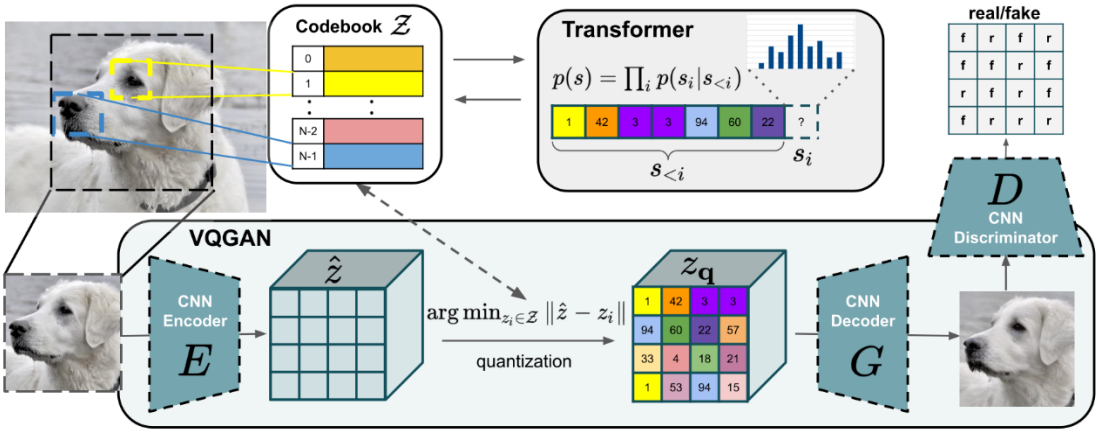


*Figure 2: VQGAN Network For DALLE[5]*

During the training process, the encoder and decoder are working together to minimize the reconstruction loss. The reconstruction loss is a measure of how well the decoder is able to reconstruct the original data from the latent representation. By minimizing the reconstruction loss, the encoder and decoder learn to represent the data in a way that is both efficient and informative.

VAEs have been used for a variety of tasks, including image generation, text generation, and drug discovery. They are a powerful tool for learning the latent representation of data, and they have the potential to be used for a wide variety of applications.

Since I need to use VAE and training it from scratch takes a lot time, a pretrained VAE which is VQGanVAE is used.

VQGAN (Vector Quantized Generative Adversarial Networks) is a model, which is derived from the GAN architecture, that focuses on image generation and offers the advantage of producing high-quality images with fewer parameters compared to other models. In VQGAN, the generator incorporates both a noise distribution and vector quantization. This combination enables VQGAN to generate images by creating a more condensed representation using a quantization process and the creation of a codebook vector [5][6]. Specifically, VQGAN is built upon the VQVAE (Vector Quantized Variational Autoencoder) architecture, which differs from traditional approaches by utilizing discrete encoder outputs rather than continuous.

In VQVAE, the prior distribution is learned instead of being fixed. The images are mapped via encoder to a discrete latent sequence, and then the images are reconstructed through decoder.

The encoder output is quantized, and in this way, it cannot be differentiated. Instead, via the decoder the reconstruction error is backpropagated, and the final gradient is found and cloned to the encoder.

## 3.RESULTS

### 3.1 DCGAN RESULTS

The results that obtained from DCGAN architecture are given below. Firstly the loss graph is depicted to see the model performance and learning process.

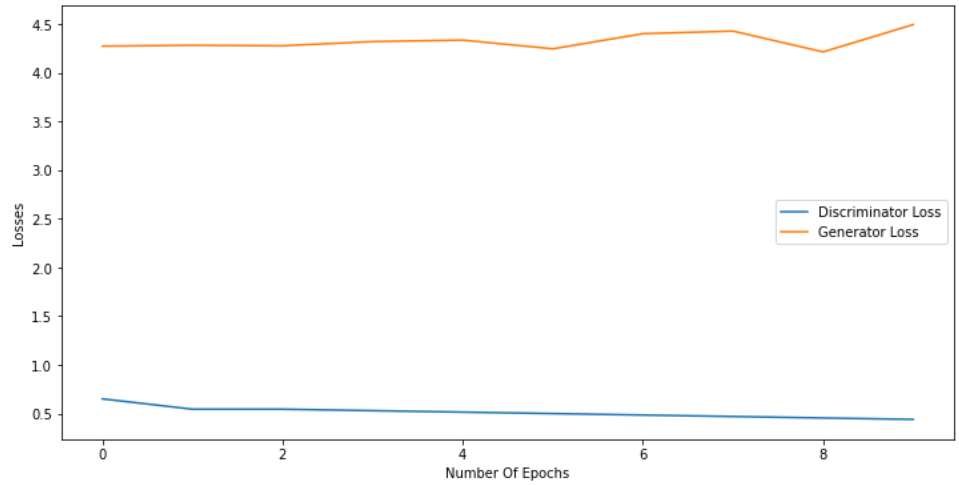


*Figure 3: Discriminator loss vs Generator loss for 10 epochs*

As seen from the figure above the loss of generator is higher than discriminator as expected. However, the increase of generator was not an expected result. This problem indicates that there were something wrong in the training process. Just looking at this images it can be predicted that the images would not be well-generated as well.

Lets examine a batch from the test set after 10 epoch.

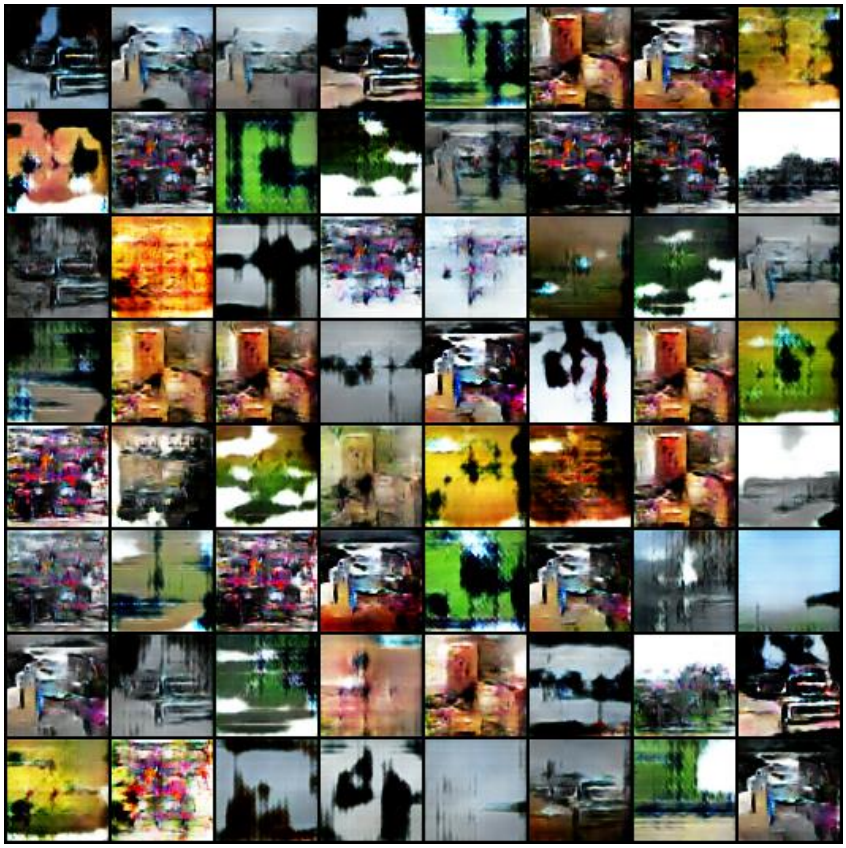

*Figure 4:Batch from test set after 10 epochs*

When this image is examined, it can be seen that none of the pictures have meaningful shapes and those images cannot be interpreted. Aditionally it can be seen that some of the images are completely identical even if the captions are different. Lets examine some of them seperately.

*Figure 5:a man is sitting in a busy area with his bike*

When we examine the caption and images it can be said that there is no relations. Even if one say there is a bike I think it is just an imagination.

Lets examine some other examples.

*Figure 6: a man is sitting in a busy area with his bike*

*Figure 7: an office desk with a lot of x_UNK_ and a x_UNK_ on computer*

*Figure 8: x_UNK_ bathroom with broken toilet next to a x_UNK_*

When the resulting images are examined it can be easily seen that, it is not a good model and cannot achieve text-to-image generation.

### 3.2 DALL-E RESULTS

In order to see the model performance firstly lest examine the loss graphs. Which are depicted below. While the first one shows the batch loss graph the other one shows the total loss for an epoch.

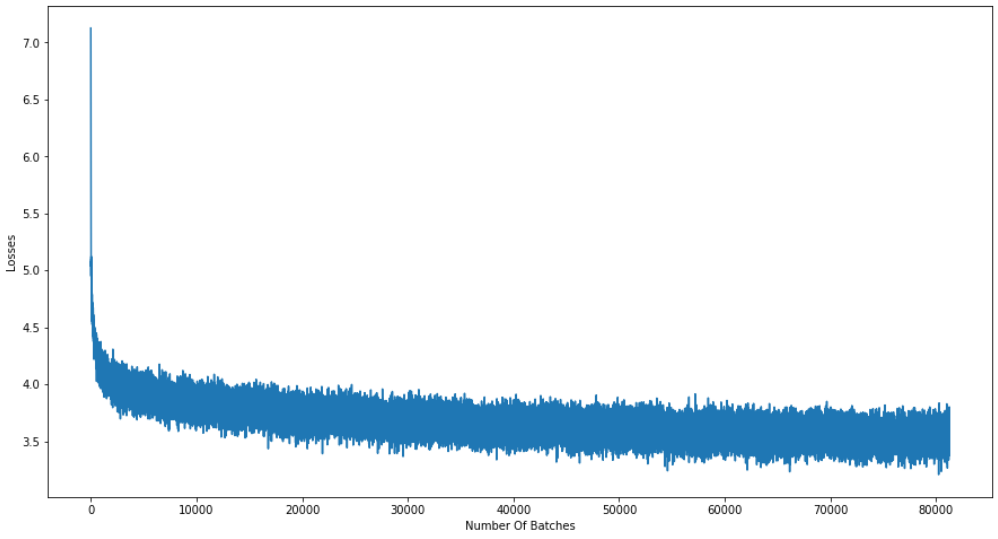


*Figure 9: Batch Loss Graph*

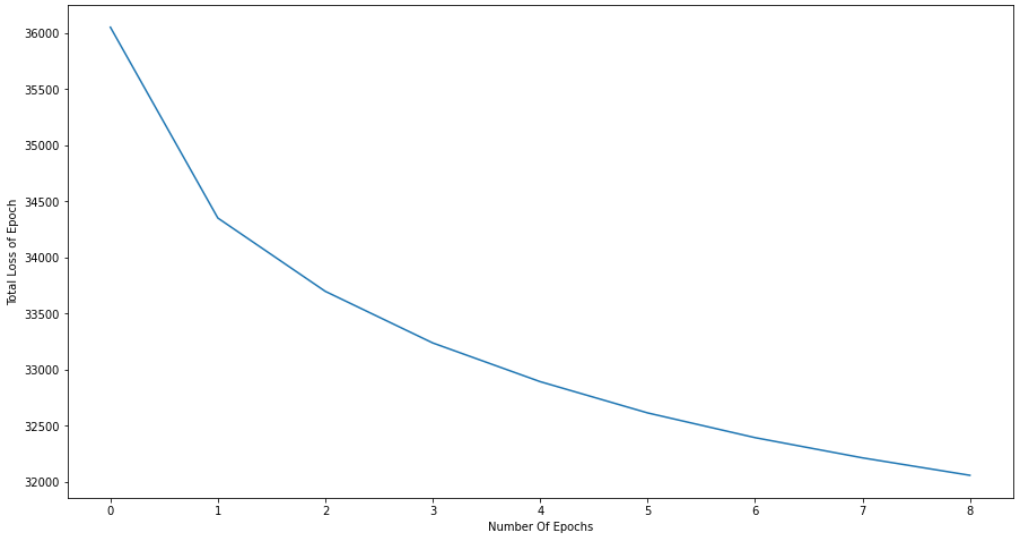


*Figure 10: Total Loss for an epoch*

Unlike DCGAN we can see that the loss is decreasing over epochs and if the total number of epochs were higher it could be possible to get much better results.

Lets examine some of the output images to see the performance of the model.

*Figure 11: two computer monitors a laptop keyboard and a mouse set up on a desk*

*Figure 12: a bathroom with a double sink and x_UNK_*

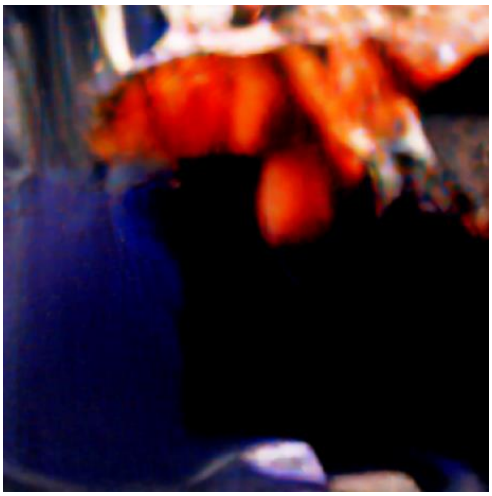

*Figure 13: a bowl of oranges near a pillow and x_UNK_x_UNK_*

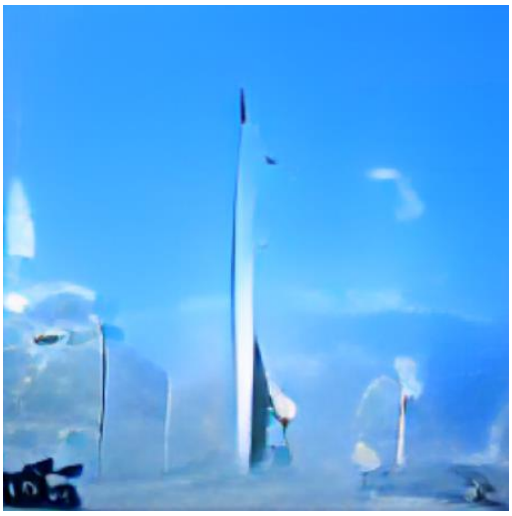

*Figure 14: an airplane carrying a space x_UNK_ in air*

*Figure 15: a street with a traffic light under a cloudy sky*

*Figure 16: a man that is standing near a plane in the grass*

As we can see from the examples, even if the generated images are not perfect, the model can understand some of the concepts and generate an image according to those words. Therefore, it is obvious that it outperform DCGAN in this task. If the total number of epochs were higher I believe the results could be much better. Even if this perforance is not perfect, I think it achieved success by creating related images to the given captions from testset.

**4. DISCUSSION**

I have started to project by obtaining the training images from provided Flickrs, but due to the presence of corrupted links, and then I have only extracted captions from available images. Then the images are preprocessed before using the models. In this process I have decided to use ImageNet mean and std because most of the pretrained models use imagenet data as input. Due to the memory limitations

I resize images to 64x64 for DCGAN while DALL-E model can handle 256x256 sized images. Through an extensive research , I determined that GANs (Generative Adversarial Networks) were well-suited for creating fake images. Consequently, I employed DCGAN and DALL-E architectures. DALL-E, incorporating a discrete VAE component, harnessed a pre-trained VQGAN model sourced from ImageNet. Then I have explored the application of VAE, GAN and various autoencoders, which we have done before for miniproject, for text-to-image synthesis. To facilitate contextual understanding, we utilized text embedding representations of input prompts, building upon our previous knowledge from the mini-project. However, due to computational limitations, I could not fully leverage the training potential of the models. Despite training each model for a few epochs, I observed ongoing enhancements during the training especially for DALLE, suggesting that further increasing the epoch count would improve model performance. Additionally, I realized that GANs were the most suitable choice for this problem, and usage of pre-trained models is much more effective than training models from scratch. This is because while I have utilized pretrained model for DALL-E, DCGAN was a from scratch model, only the embeddding is done using pretrained BERT. While the generated images did not reach hyper-realistic quality, I consider the project is successful because our primary aim was to produce realistic images and the goal has accomplished with the use of DALL-E.

On top of those I want to indicate that all of the work is my own work because my teammates didn't contrbiute to the project.

## 5.REFERENCES


[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, et al., "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27, Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[2] J. Li, J. Jia, and D. Xu, "Unsupervised representation learning of image-based plant disease with deep convolutional generative Adversarial Networks," *2018 37th Chinese Control Conference (CCC)*, 2018. doi:10.23919/chicc.2018.8482813

[3] DCGAN tutorial — PYTORCH tutorials 2.0.1+CU117 documentation, https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html (accessed Jun. 11, 2023).

[4] Tokenizer, https://huggingface.co/docs/transformers/main_classes/tokenizer (accessed Jun. 11, 2023).

[5] P. Esser, R. Rombach, and B. Ommer, "Taming transformers for high-resolution image synthesis," arXiv.org, https://arxiv.org/abs/2012.09841 (accessed Jun. 11, 2023).

[6] Lucidrains, “Lucidrains/Dalle-pytorch: Implementation / replication of dall-e, OpenAI’s text to image transformer, in Pytorch,” GitHub, https://github.com/lucidrains/DALLE-pytorch (accessed Jun. 11, 2023).

## 6. APPENDIX- CODES

All the codes and models can be accessed from the link:

https://drive.google.com/drive/folders/1FFYFnbh0epC0W0EHxiIMy3G4j3kYSWD-?usp=sharing

Also, I have used Kaggle and the data is uploaded to Kaggle, this data can be accessed from the links below:

https://www.kaggle.com/datasets/ouzcanduran/cleaned-data

https://www.kaggle.com/datasets/ouzcanduran/443-datasets

https://www.kaggle.com/datasets/ouzcanduran/443-captions

https://www.kaggle.com/datasets/ouzcanduran/443-train-images

https://www.kaggle.com/datasets/ouzcanduran/443-test-images

**#####Download_Images.ipynb**

```
import h5py
import requests
import os
import numpy as np
from tqdm.notebook import tqdm

path = r"C:\Users\Oguz\Desktop\Untitled Folder\443\\"

train_path = path + 'trainimages'
test_path= path + 'testimages'
```

```
if not os.path.exists(train_path):
    os.mkdir(train_path)

if not os.path.exists(test_path):
    os.mkdir(test_path)

train_dataset=h5py.File(r"C:\Users\Oguz\Desktop\Untitled
Folder\443\eee443_project_dataset_train.h5",'r')
test_dataset=h5py.File(r"C:\Users\Oguz\Desktop\Untitled
Folder\443\eee443_project_dataset_test.h5",'r')

train_url = train_dataset["train_url"][()]
test_url = test_dataset["test_url"][()]

def download_images(img_path, data_url):

    os.makedirs(img_path, exist_ok=True)
    os.chdir(img_path)

    for item in tqdm(range(len(data_url))):

        url = data_url[item].decode('utf-8')

        filename = str(item) + ".png"
        filepath = os.path.join(img_path, filename)

        reqs = requests.get(url, allow_redirects=True, stream=True)
        if reqs.status_code == 200:
            with open(filepath, 'wb') as f:
                for chunk in reqs.iter_content(1024):
                    f.write(chunk)
    return

if os.path.exists(train_path):
```

```
    download_images(train_path, train_url)

if os.path.exists(test_path):
    download_images(test_path, test_url)
```

**####Clear_Data.ipynb**

```
import os
import h5py
import numpy
from tqdm import tqdm

train_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_train.h5','r')
train_cap = train_dataset["train_cap"][()]
train_imid = train_dataset["train_imid"][()]
train_url = train_dataset["train_url"][()]

test_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_test.h5','r')
test_cap = test_dataset["test_caps"][()]
test_imid = test_dataset["test_imid"][()]
test_url=test_dataset["test_url"][()]

train_img_dir="/kaggle/input/443-train-images/443images"
test_img_dir="/kaggle/input/443-test-images/443test_images"

direc= os.listdir(train_img_dir)
truth_table=[]
for i in tqdm(train_imid):
    if str(i)+ ".png" in direc:
        truth_table.append(True)
    else:
        truth_table.append(False)
truth_array=np.array(truth_table)
```

```
train_imid2=train_imid[truth_array]
train_cap2=train_cap[truth_array]
np.save("trainimid.npy",train_imid2)
np.save("traicap.npy",train_cap2)

direc2= os.listdir(test_img_dir)
truth_table2=[]
for i in tqdm(test_imid):
  if str(i)+ ".png" in direc2:
    truth_table2.append(True)
  else:
    truth_table2.append(False)

truth_array2=np.array(truth_table2)
test_imid2=test_imid[truth_array2]
test_cap2=test_cap[truth_array2]
np.save("testimid.npy",test_imid2)
np.save("testcap.npy",test_cap2)
```

**####dallE.ipynb**

```
import h5py
import pandas as pd
import numpy as np
import os
from tqdm import tqdm
import requests
from PIL import Image
import time
import torch
import h5py
```

```
import glob
import cv2
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from torchvision import transforms as T
from torch.optim.lr_scheduler import ExponentialLR
from tqdm import tqdm
! pip install dalle-pytorch
!pip install pytorch-lightning==1.8.4
from dalle_pytorch import __version__
from torch.optim import Adam
from torch.optim.lr_scheduler import ExponentialLR
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid, save_image
from pathlib import Path
from dalle_pytorch import OpenAIDiscreteVAE, DiscreteVAE, DALLE, VQGanVAE
ImageFile.LOAD_TRUNCATED_IMAGES = True
from tqdm.notebook import tqdm

train_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_train.h5','r')

train_cap = train_dataset["train_cap"][()]
train_imid = train_dataset["train_imid"][()]
train_url = train_dataset["train_url"][()]
word_code = train_dataset["word_code"][()]



test_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_test.h5','r')
test_cap = test_dataset["test_caps"][()]
test_imid = test_dataset["test_imid"][()]
test_url=test_dataset["test_url"][()]
```

```
train_img_dir="/kaggle/input/443-train-images/443images"
test_img_dir="/kaggle/input/443-test-images/443test_images"

train_imid2=np.load("/kaggle/input/cleaned-data/trainimid.npy")
train_cap2=np.load("/kaggle/input/cleaned-data/traicap.npy")
test_imid2=np.load("/kaggle/input/cleaned-data/testimid.npy")
test_cap2=np.load("/kaggle/input/cleaned-data/testcap.npy")

class Create_Dataset(Dataset):

  def __init__(self, imid, cap, dir_path, **kwargs):

   self.dir_path = dir_path
   self.sample_n = len(imid)

   self.imid = imid
   self.cap = cap

  def __len__(self):

    return self.sample_n

  def __getitem__(self, ind):

     imgpath = self.dir_path + "/" + str(self.imid[ind]) + ".png"
     self.image = Image.open(imgpath).convert("RGB")

     data_tensor = self.ImageToTensor(self.image)

     return (data_tensor, self.cap[ind])

  def ImageToTensor(self, img):
```

```
    transform_with_resize = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize((0.485,0.456,0.406), (0.229,0.224,0.225))
    ])
    get_image = transforms.Compose([transforms.ToPILImage()])
    img_rsz = transform_with_resize(img)
    tns_to_img = get_image(img_rsz)

    if img_rsz.shape[0]>3:
     img_rsz = img_rsz[:3]
    return img_rsz

  def get_splits(self):

    n_tra = int(0.85 * self.sample_n)
    n_val = self.sample_n-n_tra

    self.get_train, self.get_val = torch.utils.data.random_split(self, [n_tra, n_val])

train_set = Create_Dataset(train_imid2, train_cap2, train_img_dir)
test_set = Create_Dataset(test_imid2, test_cap2, test_img_dir)
torch.cuda.empty_cache()

train_set.get_splits()
validation_data = train_set.get_val
train_data = train_set.get_train

print(len(train_data))
print(len(validation_data))

parameters=dict(
```

```
    dim = 256,
    num_text_tokens = 1004,
    text_seq_len = 17,
    depth = 2,
    heads = 8,
    dim_head = 64,
    attn_dropout = 0.05,
    ff_dropout = 0.05,
    reversible= False,
    stable= False,
    shift_tokens= False,
    rotary_emb= False,
    share_input_output_emb= False)

def model_save(path, opt,hparams, epoch=0):

    save_obj = {

            'hparams': hparams,

            'epoch': epoch,

            'version': __version__,

            'vae_class_name': "VQGanVAE",

            'weights': dall_e.state_dict(),

            'pt_opt': opt.state_dict(),

            'pt_sch': sch.state_dict() }

    torch.save(save_obj, path)
```

```
#dall_e = DALLE(**parameters,vae = VQGanVAE()).to(device) ## use it only for first epoch

pathpt="/kaggle/input/newwpoch3/dalle_ep3.pt"   ##### dont use for first epoch

ptload=torch.load(pathpt)                       ####  dont use for first epoch

pt_params=ptload["hparams"]                      ####  dont use for first epoch
pt_weights=ptload["weights"]                     ####  dont use for first epoch
pt_opt=ptload.get("pt_opt")                      ####  dont use for first epoch
pt_sch=ptload.get("pt_sch")                      ####  dont use for first epoch

dall_e=DALLE(**pt_params,vae = VQGanVAE()).to(device)  #### dont use for first epoch
dall_e.load_state_dict(pt_weights)               #### dont use for first epoch

opt = torch.optim.Adam(dall_e.parameters(), lr=0.001)
opt.load_state_dict(pt_opt)                      ####  dont use for first epoch


sch = torch.optim.lr_scheduler.ExponentialLR(opt, 0.98)
sch.load_state_dict(pt_sch)                      ####  dont use for first epoch

train_dataloader = torch.utils.data.DataLoader(
    dataset=train_data,
    batch_size=32,
    shuffle=True,
    num_workers=2
)

dall_e.train()
resume=8
num_epoch=1
for ep in range(num_epoch):
```

```
    print("Epoch:",ep)
    temp_losses=[]
    for x, y in tqdm(train_dataloader):
        print()
        opt.zero_grad()
        loss = dall_e(y.to(device), x.to(device), return_loss = True)
        loss.backward()
        opt.step()

        temp_losses.append(loss.item())

        print("Batch Loss: " + str(loss.item()))
    sch.step()

import os
import json
from kaggle_secrets import UserSecretsClient

secrets = UserSecretsClient()

os.environ['KAGGLE_USERNAME'] = secrets.get_secret("KAGGLE_USERNAME")
os.environ['KAGGLE_KEY'] = secrets.get_secret("KAGGLE_KEY")

os.makedirs('/kaggle/dataset/', exist_ok=True)
meta = dict(
    id="ouzcanduran/my-dataset9",
    title="Dalle_ep10",
    isPrivate=True,
    licenses=[dict(name="other")]
)
with open('/kaggle/dataset/dataset-metadata.json', 'w') as f:
    json.dump(meta, f)
```

os.makedirs('/kaggle/dataset/my-directory', exist_ok=True)

model_save('/kaggle/dataset/my-directory/dalle_ep{}.pt'.format(resume+1), opt,parameters, epoch=resume+1)

np.save("/kaggle/dataset/my-directory/losses_9ep.npy",np.array(temp_losses))

!kaggle datasets create -p "/kaggle/dataset" --dir-mode zip

**####dalletest.ipynb**

import h5py

import pandas as pd

import numpy as np

import os

from tqdm import tqdm

import requests

from PIL import Image

import time

import torch

import h5py

import glob

import cv2

import matplotlib.pyplot as plt

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms

from torchvision import transforms as T

from torch.optim.lr_scheduler import ExponentialLR

from tqdm import tqdm

! pip install dalle-pytorch

!pip install pytorch-lightning==1.8.4

from dalle_pytorch import __version__

```
from torch.optim import Adam
from torch.optim.lr_scheduler import ExponentialLR
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid, save_image
from pathlib import Path
from dalle_pytorch import OpenAIDiscreteVAE, DiscreteVAE, DALLE, VQGanVAE
ImageFile.LOAD_TRUNCATED_IMAGES = True
from tqdm.notebook import tqdm

train_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_train.h5','r')

train_cap = train_dataset["train_cap"][()]
train_imid = train_dataset["train_imid"][()]
train_url = train_dataset["train_url"][()]
word_code = train_dataset["word_code"][()]



test_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_test.h5','r')
test_cap = test_dataset["test_caps"][()]
test_imid = test_dataset["test_imid"][()]
test_url=test_dataset["test_url"][()]

train_img_dir="/kaggle/input/443-train-images/443images"
test_img_dir="/kaggle/input/443-test-images/443test_images"



train_imid2=np.load("/kaggle/input/cleaned-data/trainimid.npy")
train_cap2=np.load("/kaggle/input/cleaned-data/traicap.npy")
test_imid2=np.load("/kaggle/input/cleaned-data/testimid.npy")
test_cap2=np.load("/kaggle/input/cleaned-data/testcap.npy")

class Create_Dataset(Dataset):
```

```
def __init__(self, imid, cap, dir_path, **kwargs):

 self.dir_path = dir_path
 self.sample_n = len(imid)

 self.imid = imid
 self.cap = cap

def __len__(self):

  return self.sample_n

def __getitem__(self, ind):

   imgpath = self.dir_path + "/" + str(self.imid[ind]) + ".png"
   self.image = Image.open(imgpath).convert("RGB")

   data_tensor = self.ImageToTensor(self.image)

   return (data_tensor, self.cap[ind])

def ImageToTensor(self, img):

  transform_with_resize = transforms.Compose([
      transforms.Resize((256, 256)),
      transforms.ToTensor(),
      transforms.Normalize((0.485,0.456,0.406), (0.229,0.224,0.225))
  ])
  get_image = transforms.Compose([transforms.ToPILImage()])
  img_rsz = transform_with_resize(img)
  tns_to_img = get_image(img_rsz)
```

```
    if img_rsz.shape[0]>3:
     img_rsz = img_rsz[:3]
    return img_rsz

  def get_splits(self):

    n_tra = int(0.85 * self.sample_n)
    n_val = self.sample_n-n_tra

    self.get_train, self.get_val = torch.utils.data.random_split(self, [n_tra, n_val])

train_set = Create_Dataset(train_imid2, train_cap2, train_img_dir)
test_set = Create_Dataset(test_imid2, test_cap2, test_img_dir)
torch.cuda.empty_cache()

train_set.get_splits()
validation_data = train_set.get_val
train_data = train_set.get_train

print(len(train_data))
print(len(validation_data))

parameters=dict(
    dim = 256,
    num_text_tokens = 1004,
    text_seq_len = 17,
    depth = 2,
    heads = 8,
    dim_head = 64,
    attn_dropout = 0.05,
    ff_dropout = 0.05,
    reversible= False,
    stable= False,
```

```
    shift_tokens= False,
    rotary_emb= False,
    share_input_output_emb= False)

def model_save(path, opt,hparams, epoch=0):

    save_obj = {

            'hparams': hparams,

            'epoch': epoch,

            'version': __version__,

            'vae_class_name': "VQGanVAE",

            'weights': dall_e.state_dict(),

            'pt_opt': opt.state_dict(),

            'pt_sch': sch.state_dict() }

    torch.save(save_obj, path)

#dall_e = DALLE(**parameters,vae = VQGanVAE()).to(device) ## use it only for first epoch

pathpt="/kaggle/input/newwpoch3/dalle_ep3.pt"   ##### dont use for first epoch

ptload=torch.load(pathpt)                       ####  dont use for first epoch

pt_params=ptload["hparams"]                      ####  dont use for first epoch
pt_weights=ptload["weights"]                     ####  dont use for first epoch
pt_opt=ptload.get("pt_opt")                      ####  dont use for first epoch
```

```
pt_sch=ptload.get("pt_sch")                          ####  dont use for first epoch

dall_e=DALLE(**pt_params,vae = VQGanVAE()).to(device)  #### dont use for first epoch
dall_e.load_state_dict(pt_weights)                   #### dont use for first epoch

opt = torch.optim.Adam(dall_e.parameters(), lr=0.001)
opt.load_state_dict(pt_opt)                          ####  dont use for first epoch


sch = torch.optim.lr_scheduler.ExponentialLR(opt, 0.98)
sch.load_state_dict(pt_sch)                          ####  dont use for first epoch

words = pd.DataFrame(word_code)
words = words.sort_values(0, axis=1)
array_of_words = np.asarray(words.columns)
word_dict = {}
index_dict = {}

for i in range(len(array_of_words)):
 w = array_of_words[i]
 word_dict[w] = i
 index_dict[i] = w

index=443 ##  change it for different captions

capt = torch.tensor(test_cap2[index]).unsqueeze(dim=0).to(device)

images = dall_e.generate_images(capt)

plt.axis("off")
plt.imshow( images.cpu().detach().squeeze().permute(1, 2, 0) )
```

```
print(test_url[test_imid2[index]])
sentence=[]

 for i in test_cap2[index]:

 sentence.append(index_dict[i])

print(sentence)
```

**####dcgaN.ipynb**

```
import h5py
import pandas as pd
import numpy as np
import os
from tqdm import tqdm
import requests
from PIL import Image
import time
import torch
import h5py
import glob
import cv2
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from transformers import BertTokenizer, TFBertModel
import torchtext.vocab as vocab
import argparse
```

```
import random
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from torch.utils.data import Dataset
from torchvision import transforms
from transformers import BertTokenizer, TFBertModel
from tqdm.notebook import tqdm
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

train_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_train.h5','r')

train_cap = train_dataset["train_cap"][()]
train_imid = train_dataset["train_imid"][()]
train_url = train_dataset["train_url"][()]
train_ims = train_dataset["train_ims"][()]
word_code = train_dataset["word_code"][()]



test_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_test.h5','r')
test_cap = test_dataset["test_caps"][()]
test_imid = test_dataset["test_imid"][()]
test_url=test_dataset["test_url"][()]
```

```
words = pd.DataFrame(word_code)
words = words.sort_values(0, axis=1)
array_of_words = np.asarray(words.columns)

word_dict = {}
index_dict = {}
for i in range(len(array_of_words)):
 w = array_of_words[i]
 word_dict[w] = i
 index_dict[i] = w

train_img_dir="/kaggle/input/443-train-images/443images"
test_img_dir="/kaggle/input/443-test-images/443test_images"

train_imid2=np.load("/kaggle/input/cleaned-data/trainimid.npy")
train_cap2=np.load("/kaggle/input/cleaned-data/traicap.npy")
test_imid2=np.load("/kaggle/input/cleaned-data/testimid.npy")
test_cap2=np.load("/kaggle/input/cleaned-data/testcap.npy")

class Create_Dataset(Dataset):

  def __init__(self, imid, cap, dir_path, **kwargs):

   self.dir_path = dir_path
   self.sample_n = len(imid)

   self.imid = imid
   self.cap = cap

  def __len__(self):

    return self.sample_n
```

```
  def __getitem__(self, ind):

    imgpath = self.dir_path + "/" + str(self.imid[ind]) + ".png"
    self.image = Image.open(imgpath).convert("RGB")

    data_tensor = self.ImageToTensor(self.image)

    return (data_tensor, self.cap[ind])

  def ImageToTensor(self, img):

    transform_with_resize = transforms.Compose([
        transforms.Resize(64),
        transforms.CenterCrop(64),
        transforms.ToTensor(),
        transforms.Normalize((0.485,0.456,0.406), (0.229,0.224,0.225))
    ])
    get_image = transforms.Compose([transforms.ToPILImage()])
    img_rsz = transform_with_resize(img)
    tns_to_img = get_image(img_rsz)

    if img_rsz.shape[0]>3:
     img_rsz = img_rsz[:3]
    return img_rsz

  def get_splits(self):

    n_tra = int(0.85 * self.sample_n)
    n_val = self.sample_n-n_tra

    self.get_train, self.get_val = torch.utils.data.random_split(self, [n_tra, n_val])

train_set = Create_Dataset(train_imid2, train_cap2, train_img_dir)
```

```
test_set = Create_Dataset(test_imid2, test_cap2, test_img_dir)
torch.cuda.empty_cache()

train_set.get_splits()
validation_data = train_set.get_val
train_data = train_set.get_train
print(len(train_data))
print(len(validation_data))

torch.cuda.empty_cache()
train_dataloader = torch.utils.data.DataLoader(

  dataset=train_data,

  batch_size=64,

  shuffle=True,

  num_workers=0,

  drop_last = True

)

torch.cuda.empty_cache()
val_dataloader = torch.utils.data.DataLoader(

    dataset=validation_data,

    batch_size=64,

    shuffle=False,
```

```
    num_workers=0,
    drop_last = True
)
torch.cuda.empty_cache()
test_dataloader = torch.utils.data.DataLoader(
    dataset=test_set,
    batch_size=64,
    shuffle=False,
    num_workers=0,
    drop_last = True
)
print(len(test_dataloader))
name_bert="prajjwal1/bert-small"
tokenizer=BertTokenizer.from_pretrained(name_bert)
model = TFBertModel.from_pretrained(name_bert,from_pt=True)
def enc_sntcs(sntc):
 cap_list = []
```

```
for i in sntc:
    cap_list.append(array_of_words[i])
s = " ".join(cap_list)


encd = tokenizer.batch_enc_plus(
            [s],
            max_length=512,
            pad_to_max_length=False,
            return_tensors="pt",
            truncation=True,
        )


input_ids = np.array(encd["input_ids"], dtype="int32")
out = model(input_ids)
seq, pool = out[:2]
return pool[0]
def enc_sntcs_batch(sntcs):
```

```
 enc_sntcs = []
 for sntc in sntcs:
  enc_sntcs.append(enc_sntcs(sntc))
 return torch.tensor(np.array(enc_sntcs))

rand_index=443 ## for reproduction
random.seed(rand_index)
torch.manual_seed(rand_index)

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

workers = 1
batch_size = 64
image_size = 64
nc = 3
nz = 128
ngf = 64
ndf = 64
num_epochs = 1
lr = 2e-4
beta1 = 0.5
ngpu = 2

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

def weights_init(m):

  classname = m.__class__.__name__

  if classname.find('Conv') != -1:
    nn.init.xavier_uniform_(m.weight.data)

  elif classname.find("Linear") != -1:
```

```
    nn.init.xavier_uniform_(m.weight.data)
    nn.init.constant_(m.bias.data, 0)

  elif classname.find('BatchNorm') != -1:
    nn.init.normal_(m.weight.data, 1.0, 0.02)
    nn.init.constant_(m.bias.data, 0)

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

class Generator(nn.Module):
  def __init__(self,ngpu):
    super(Generator,self).__init__()
    self.ngpu = ngpu
    self.proj_dim_emb = 128
    self.noise_dim = nz
    self.latent_dim = self.noise_dim + self.proj_dim_emb
    self.epsilon = torch.randn(batch_size,128,1,1,device=device)
    self.project = nn.Sequential (
                nn.Linear(512,self.proj_dim_emb),
                nn.BatchNorm1d(num_features=self.proj_dim_emb),
                nn.LeakyReLU(negative_slope=0.2,inplace = True))
    self.main = nn.Sequential(
      nn.ConvTranspose2d(self.latent_dim, ngf * 8, 4, 1, 0, bias=False),
      nn.BatchNorm2d(ngf * 8),
      nn.ReLU(True),
      nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 4),
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 2),
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf),
```

```
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
      nn.Tanh()
      )

  def forward(self, inp,n):

    emb = self.project(inp).unsqueeze(2).unsqueeze(3)
    lat = torch.concat([emb,n],1)

    return self.main(lat)

netG = Generator(ngpu).to(device)
if (device.type == 'cuda') and (ngpu > 1):
  netG = nn.DataParallel(netG, list(range(ngpu)))
# netG.apply(weights_init)  ## Use it for first epoch
netG.load_state_dict(torch.load("/kaggle/input/dcganep1/netG_0_bs_64.pth")) ## for the next epochs

class Discriminator(nn.Module):
  def __init__(self, ngpu):
    super(Discriminator, self).__init__()
    self.ngpu = ngpu
    self.dim_emb = 512
    self.proj_dim_emb = 128
    self.main = nn.Sequential(
      nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ndf * 2),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ndf * 4),
      nn.LeakyReLU(0.2, inplace=True),
```

```
      nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ndf * 8),
      nn.LeakyReLU(0.2, inplace=True)   )

    self.proj = nn.Sequential(
        nn.Linear(self.dim_emb,self.proj_dim_emb),
        nn.BatchNorm1d(self.proj_dim_emb),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)   )

    self.aug = nn.Sequential(
        nn.Conv2d(8*ndf  + self.proj_dim_emb, 1, 4, 1, 0, bias=False),
        nn.Sigmoid()  )



  def forward(self, inpimg,capt):
    intimg= self.main(inpimg)
    emb_proj = self.proj(capt)
    emb_rep = emb_proj.repeat(4,4,1,1).permute(2,3,0,1)
    conc_hid = torch.concat([intimg,emb_rep],1)
    out = self.aug(conc_hid)
    return out.view(-1,1).squeeze(1)

netD = Discriminator(ngpu).to(device)
if (device.type == 'cuda') and (ngpu > 1):
 netD = nn.DataParallel(netD, list(range(ngpu)))
#netD.apply(weights_init)  ## Use for first epoch
netD.load_state_dict(torch.load("/kaggle/input/dcganep1/netD_0_bs_64.pth")) ## for the next epochs

criterion = nn.BCELoss()
fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device)
real_label = 1.
fake_label = 0.
```

```
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# Commented out IPython magic to ensure Python compatibility.
### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

img_list = []
G_losses = []
D_losses = []
G_losses_val = []
D_losses_val = []
iters = 0



for epoch in range(num_epochs):
  i=0

  for imgs, txt_emb in tqdm(train_dataloader):

    netD.zero_grad()
    img = imgs.to(device)
    capts = enc_sntcs_batch(txt_emb).to(device)
    b_size = img.size(0)
    label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
    capts = capts.to(torch.float32)
    output = netD(img,capts).view(-1)
    errD_real = criterion(output, label)
    errD_real.backward()
    D_x = output.mean().item()
    noise = torch.randn(b_size, nz,1,1, device=device)
    fake = netG(capts,noise)
    label.fill_(fake_label)
    output = netD(fake.detach(),capts).view(-1)
```

```
    errD_fake = criterion(output, label)
    errD_fake.backward()
    D_G_z1 = output.mean().item()
    errD = errD_real + errD_fake
    optimizerD.step()

    netG.zero_grad()
    label.fill_(real_label)
    output = netD(fake,capts).view(-1)
    errG = criterion(output, label)
    errG.backward()

    D_G_z2 = output.mean().item()
    optimizerG.step()

    if i % 50 == 0:
        print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
#           % (epoch, num_epochs, i, len(train_dataloader), errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))


    G_losses.append(errG.item())
    D_losses.append(errD.item())

    if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(train_dataloader)-1)):
        with torch.no_grad():
            fake = netG(capts,fixed_noise).detach().cpu()
        img_list.append(vutils.make_grid(fake, padding=2, normalize=True))
    i += 1
    iters += 1


  torch.save(netG.state_dict(),"/kaggle/working/netG_1_bs.pth")
  torch.save(netD.state_dict(),"/kaggle/working/netD_1_bs.pth")
```

```
np.save("gloss1.npy",np.array(G_losses))
np.save("dloss1.npy",np.array(D_losses))

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())

real_batch = next(iter(test_dataloader))
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
                        padding=5, normalize=True).cpu(),(1,2,0)))
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
```

```
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

**####dcgantest.ipynb**

```
import h5py
import pandas as pd
import numpy as np
import os
from tqdm import tqdm
import requests
from PIL import Image
import time
import torch
import h5py
import glob
import cv2
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from transformers import BertTokenizer, TFBertModel
import torchtext.vocab as vocab
import argparse
import random
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
```

```
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from torch.utils.data import Dataset
from torchvision import transforms
from transformers import BertTokenizer, TFBertModel
from tqdm.notebook import tqdm
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from PIL import ImageFile
from torchvision.utils import save_image,make_grid
ImageFile.LOAD_TRUNCATED_IMAGES = True

train_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_train.h5','r')

train_cap = train_dataset["train_cap"][()]
train_imid = train_dataset["train_imid"][()]
train_url = train_dataset["train_url"][()]
train_ims = train_dataset["train_ims"][()]
word_code = train_dataset["word_code"][()]



test_dataset = h5py.File('/kaggle/input/443-datasets/eee443_project_dataset_test.h5','r')
test_cap = test_dataset["test_caps"][()]
test_imid = test_dataset["test_imid"][()]
test_url=test_dataset["test_url"][()]

words = pd.DataFrame(word_code)
```

```
words = words.sort_values(0, axis=1)
array_of_words = np.asarray(words.columns)

word_dict = {}
index_dict = {}
for i in range(len(array_of_words)):
 w = array_of_words[i]
 word_dict[w] = i
 index_dict[i] = w

train_img_dir="/kaggle/input/443-train-images/443images"
test_img_dir="/kaggle/input/443-test-images/443test_images"

train_imid2=np.load("/kaggle/input/cleaned-data/trainimid.npy")
train_cap2=np.load("/kaggle/input/cleaned-data/traicap.npy")
test_imid2=np.load("/kaggle/input/cleaned-data/testimid.npy")
test_cap2=np.load("/kaggle/input/cleaned-data/testcap.npy")

class Create_Dataset(Dataset):

  def __init__(self, imid, cap, dir_path, **kwargs):

   self.dir_path = dir_path
   self.sample_n = len(imid)

   self.imid = imid
   self.cap = cap

  def __len__(self):

    return self.sample_n

  def __getitem__(self, ind):
```

```
    imgpath = self.dir_path + "/" + str(self.imid[ind]) + ".png"
    self.image = Image.open(imgpath).convert("RGB")

    data_tensor = self.ImageToTensor(self.image)

    return (data_tensor, self.cap[ind])

  def ImageToTensor(self, img):

   transform_with_resize = transforms.Compose([
       transforms.Resize(64),
       transforms.CenterCrop(64),
       transforms.ToTensor(),
       transforms.Normalize((0.485,0.456,0.406), (0.229,0.224,0.225))
   ])
   get_image = transforms.Compose([transforms.ToPILImage()])
   img_rsz = transform_with_resize(img)
   tns_to_img = get_image(img_rsz)

   if img_rsz.shape[0]>3:
    img_rsz = img_rsz[:3]
   return img_rsz

  def get_splits(self):

   n_tra = int(0.85 * self.sample_n)
   n_val = self.sample_n-n_tra

   self.get_train, self.get_val = torch.utils.data.random_split(self, [n_tra, n_val])

train_set = Create_Dataset(train_imid2, train_cap2, train_img_dir)
test_set = Create_Dataset(test_imid2, test_cap2, test_img_dir)
```

```
torch.cuda.empty_cache()

train_set.get_splits()
validation_data = train_set.get_val
train_data = train_set.get_train
print(len(train_data))
print(len(validation_data))

torch.cuda.empty_cache()
train_dataloader = torch.utils.data.DataLoader(

  dataset=train_data,

  batch_size=64,

  shuffle=True,

  num_workers=0,

  drop_last = True

)

torch.cuda.empty_cache()
val_dataloader = torch.utils.data.DataLoader(

    dataset=validation_data,

    batch_size=64,

    shuffle=False,

    num_workers=0,
```

```
    drop_last = True
)
torch.cuda.empty_cache()
test_dataloader = torch.utils.data.DataLoader(
    dataset=test_set,
    batch_size=64,
    shuffle=False,
    num_workers=0,
    drop_last = True
)
print(len(test_dataloader))
name_bert="prajjwal1/bert-small"
tokenizer=BertTokenizer.from_pretrained(name_bert)
model = TFBertModel.from_pretrained(name_bert,from_pt=True)
def enc_sntcs(sntc):
 cap_list = []
 for i in sntc:
```

```
    cap_list.append(array_of_words[i])
s = " ".join(cap_list)


encd = tokenizer.batch_enc_plus(
            [s],
            max_length=512,
            pad_to_max_length=False,
            return_tensors="pt",
            truncation=True,
        )


input_ids = np.array(encd["input_ids"], dtype="int32")
out = model(input_ids)
seq, pool = out[:2]
return pool[0]
def enc_sntcs_batch(sntcs):
enc_sntcs = []
```

```
 for sntc in sntcs:
  enc_sntcs.append(enc_sntcs(sntc))
 return torch.tensor(np.array(enc_sntcs))

def dec_sntcs(sntcs):
 dec_sntcs = []
 for sntc in sntcs:
    dec_sntcs.append(enc_sntcs(sntc))
    list_capt = []
    for i in sntc:
     list_capt.append(array_of_words[i])
    s = " ".join(list_capt)
    dec_sntcs.append(s)
 return dec_sntcs

rand_index=443 ## for reproduction
random.seed(rand_index)
torch.manual_seed(rand_index)

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

workers = 1
batch_size = 64
image_size = 64
nc = 3
nz = 128
ngf = 64
ndf = 64
num_epochs = 1
lr = 2e-4
beta1 = 0.5
ngpu = 2
```

```
### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

def weights_init(m):

  classname = m.__class__.__name__

  if classname.find('Conv') != -1:
    nn.init.xavier_uniform_(m.weight.data)

  elif classname.find("Linear") != -1:
    nn.init.xavier_uniform_(m.weight.data)
    nn.init.constant_(m.bias.data, 0)

  elif classname.find('BatchNorm') != -1:
    nn.init.normal_(m.weight.data, 1.0, 0.02)
    nn.init.constant_(m.bias.data, 0)

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

class Generator(nn.Module):
  def __init__(self,ngpu):
    super(Generator,self).__init__()
    self.ngpu = ngpu
    self.proj_dim_emb = 128
    self.noise_dim = nz
    self.latent_dim = self.noise_dim + self.proj_dim_emb
    self.epsilon = torch.randn(batch_size,128,1,1,device=device)
    self.project = nn.Sequential (
                nn.Linear(512,self.proj_dim_emb),
                nn.BatchNorm1d(num_features=self.proj_dim_emb),
                nn.LeakyReLU(negative_slope=0.2,inplace = True))
    self.main = nn.Sequential(
      nn.ConvTranspose2d(self.latent_dim, ngf * 8, 4, 1, 0, bias=False),
```

```
      nn.BatchNorm2d(ngf * 8),
      nn.ReLU(True),
      nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 4),
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 2),
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf),
      nn.ReLU(True),
      nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
      nn.Tanh()
      )

  def forward(self, inp,n):

    emb = self.project(inp).unsqueeze(2).unsqueeze(3)
    lat = torch.concat([emb,n],1)

    return self.main(lat)

netG = Generator(ngpu).to(device)
if (device.type == 'cuda') and (ngpu > 1):
  netG = nn.DataParallel(netG, list(range(ngpu)))

netG.load_state_dict(torch.load("/kaggle/input/dcganep1/netG_0_bs_64.pth"))

class Discriminator(nn.Module):
  def __init__(self, ngpu):
    super(Discriminator, self).__init__()
    self.ngpu = ngpu
    self.dim_emb = 512
```

```
  self.proj_dim_emb = 128
  self.main = nn.Sequential(
    nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
    nn.BatchNorm2d(ndf * 2),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
    nn.BatchNorm2d(ndf * 4),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
    nn.BatchNorm2d(ndf * 8),
    nn.LeakyReLU(0.2, inplace=True)   )

  self.proj = nn.Sequential(
      nn.Linear(self.dim_emb,self.proj_dim_emb),
      nn.BatchNorm1d(self.proj_dim_emb),
      nn.LeakyReLU(negative_slope=0.2, inplace=True)   )

  self.aug = nn.Sequential(
      nn.Conv2d(8*ndf  + self.proj_dim_emb, 1, 4, 1, 0, bias=False),
      nn.Sigmoid()  )



def forward(self, inpimg,capt):
  intimg= self.main(inpimg)
  emb_proj = self.proj(capt)
  emb_rep = emb_proj.repeat(4,4,1,1).permute(2,3,0,1)
  conc_hid = torch.concat([intimg,emb_rep],1)
  out = self.aug(conc_hid)
  return out.view(-1,1).squeeze(1)
```

```
netD = Discriminator(ngpu).to(device)
if (device.type == 'cuda') and (ngpu > 1):
 netD = nn.DataParallel(netD, list(range(ngpu)))
netD.load_state_dict(torch.load("/kaggle/input/dcganep1/netD_0_bs_64.pth"))

criterion = nn.BCELoss()
fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device)
real_label = 1.
fake_label = 0.
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

img_list = []
G_losses = []
D_losses = []
G_losses_val = []
D_losses_val = []
iters = 0



i=0

for imgs, txt_emb in tqdm(test_dataloader):
   img = imgs.to(device)
   capts = enc_sntcs_batch(txt_emb).to(device)
   b_size = img.size(0)

   dec_cap = dec_sntcs(txt_emb)
   with torch.no_grad():
    generated = netG(capts,fixed_noise).detach().cpu()
```

```
    save_image(generated, '/kaggle/dataset/my-directory/generated_samples_%d.png' % (i), normalize=True)
    img_list.append(vutils.make_grid(generated, padding=2, normalize=True))
    np.save('/kaggle/dataset/my-directory/test_captss_{}.npy'.format(i),np.array(dec_cap))
    i += 1
    if i == 40:
     break

### https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())

real_batch = next(iter(test_dataloader))
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
                        padding=5, normalize=True).cpu(),(1,2,0)))
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```