Project Report

Deep Learning

[CSE 4007]

Text-to-Image Synthesis using DC-GAN

By

Godavarthi Sai Nikhil

210214

Anish Borkar

210220

Department of Computer Science and Engineering

School of Engineering and Technology

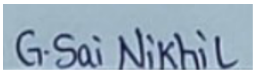
BML Munjal University

November 2023

Declaration by Candidates

We hereby declare that the project entitled “Text-to-Image Synthesis using DC-GAN” has been carried out to fulfill the partial requirements for completion of the core-elective course on Deep Learning offered in the 5 th Semester of the Bachelor of Technology (B.Tech) program in the Department of Computer Science and Engineering during AY-2023-24 (odd semester). This experimental work has been carried out by us and submitted to the course instructor Dr. Soharab Hossain Shaikh. Due acknowledgments have been made in the text of the project to all other materials used. This project has been prepared in full compliance with the requirements and constraints of the prescribed curriculum.

Name: Godavarthi Sai Nikhil

Signature of Student-1: 

Name: Anish Borkar

Signature of Student-2: 

Place: BML Munjal University

Date: 18th November 2023

Table of contents

Introduction

Text-to-Image synthesis comes under Computer Vision and NLP. It involves converting a text input into a representation, say, a latent or feature vector and using this representation to generate an image matching the textual description.

GANs, or Generative Adversarial Networks, are an approach to generative modeling using Deep Learning methods. The GAN framework consists of two entities: Generator and Discriminator. The generator produces images and tries to trick the discriminator that the generated images are real. The discriminator's job is to catch or detect the fake images generated. GANs are one of the best ways for the task of Text-to-Image synthesis. The quality of the images generated by GANs is better than one of the other models. There are many types of GANs implemented for this task, for instance, DC-GAN, RC-GAN, AttnGAN, etc. For this report's purpose, we went ahead with DC-GAN (Deep Convolutional Generative Adversarial Networks). Our contribution in this work is creating an effective GAN architecture and training process. We used the Oxford-102 Flowers datasets.

Problem Statement

Developing a Text-to-Image synthesis model using DC-GAN. The GAN must be trained on a dataset. The model should capture the semantic meaning of text descriptions, or prompt, and generate realistic images. The model should handle a variety of text descriptions.

Literature Review

The authors[1] proposed a method called LeicaGAN, consisting of a textual-visual co-embedding network (TVE), a multiple priors aggregation network (MPA) and a cascaded attentive generator (CAG). The text encoder was a pre-trained Bi-directional LSTM and the visual encoder was built upon the Inception-v3 model. The MPA network fused the sentence level embeddings. This acted as an input in the CAG, where an attention block, two residual blocks, an upsampling block and a convolution layer make up the generator. Word and Sentence-context features were produced. Two adversarial losses were employed: a visual realism adversarial loss to ensure that the generators generate visually realistic images and a text-image pair-aware adversarial loss to guarantee the semantic consistency between the input text and the generated image. For effectiveness LeicaGAN was compared with AttnGAN. CUB and Oxford-102 datasets were used and evaluation was done based on the Inception Score. LeicaGAN outperformed AttnGAN, on both the datasets.

The authors[2] proposed ControlGAN. For this, they introduced a word-level spatial and channel-wise attention-driven generator that could disentangle different visual attributes. Also, they proposed a word-level discriminator. The backbone architecture they used was AttnGAN and the text encoder was a pre-trained bi directional RNN. Conditioning Augmentation was applied. The generator exploited the attention mechanism via incorporating a spatial attention module and the channel-wise attention module. The spatial attention module dealt with words with individual spatial locations. The model was experimented on CUB and MS COCO datasets. The model proposed was compared with AttnGAN and StackGAN++ and the performance metrics were Inception Score, R-precision and L2 Error. ControlGAN gave the best results among the three for the CUB dataset.(Inception Score = 4.58 ± 0.09, Top-1 Acc(%) = 69.33 ± 3.23, L2 error = 0.18). For the COCO dataset, AttnGAN was the best in Inception Score and Top-1 Acc(%), but ControlGAN had the lowest L2 error, i.e., 0.17.

The authors[3] proposed the Self-Attention Generative Adversarial Network (SAGAN). This compared to convolutional GANs, helps with modeling long range, multi-level dependencies across image regions. Due to the self-attention, the generator can draw images in which fine details at every location are carefully coordinated

with fine details in distant portions of the image. Moreover, the discriminator can also more accurately enforce complicated geometric constraints on the global image structure. Spectral Normalisation was used in the generator and discriminator. Spectral normalization in the generator can prevent the escalation of parameter magnitudes and avoid unusual gradients. The experiment was done on the ILSVRC 2012 dataset. The evaluation metrics chosen were Inception Score and Fréchet Inception distance. The model was compared with AC-GAN and SNGAN-projection, in which SAGAN performed best, having Inception score of 52.52 and Fréchet Inception distance of 18.65.

The authors[4] proposed a StackGAN model. The model is built in two stages: Stage 1 GAN giving Low Resolution images and Stage 2 GAN giving High resolution images. The model first processes the text input and generates corresponding text embeddings to feed into the generative adversarial networks. The model included a text encoder and decoder implemented with a word-level bidirectional recurrent neural network (RNN) consisting of two long short-term memory (LSTM). The generator and discriminator receive a conditioning variable. Dataset used was COCO. The pre-trained StackGAN model has decent performance on generating images from a text input that is similar to its training set, although, when the input contains multiple objects, StackGan fails to generate the correct number of instances with clear boundaries and spatial relationships.

The authors[5] implemented DC-GAN conditioned on text features encoded by a hybrid character-level convolutional recurrent neural network. In the generator, a text query was encoded. The description embedding was compressed using a fully connected layer followed by LeakyReLU as the activation function. This was then concatenated with the noise. The discriminator consisted of several layers of strides-2 convolution with spatial batch normalization followed by LeakyReLU. The experiment was done on the CUB and Oxford-102 datasets. The GAN baseline was compared with GAN-CLS with image-text matching discriminator, GAN-INT learned with text manifold interpolation and GAN-INT-CLS which combined both.

This paper[6] described a transformer trained to autoregressive model the text and image tokens as a single stream of data. A two-stage training procedure was used: Training a discrete Variational Autoencoder to compress an 256 x 256 image into a 32 x 32 grid of image tokens and concatenating up to 256 BPE-encoded text tokens with the $32 \times 32 = 1024$ image tokens and train an autoregressive transformer to model the joint distribution over the text and image tokens. For a text-image pair, the lowercase caption is BPE-encoded using at most 256 tokens with vocabulary size 16,384. The image is encoded using $32 \times 32 = 1024$ tokens with vocabulary size 8192. The image tokens are obtained using argmax sampling from the dVAE encoder logits. The text and image tokens are concatenated and modeled autoregressive as a single stream of data. The experiment was carried out of Conceptual Captions, an extension of MS COCO. The model is compared with AttnGAN, DM-GAN and DF-GAN. The evaluation metrics were Inception Score and Fréchet Inception Distance. The zero-shot model obtained an FID score on MS-COCO within 2 points of the best prior approach, despite having never been trained on the captions. But, the model fares significantly worse on the CUB dataset, for which there is a nearly 40-point gap in FID between it and the leading prior approach, i.e., DM-GAN.

The authors[7] implemented a GAN-CLS including a generator and discriminator. The inputs were batches of images and the matching text. Both the matching and mismatching text description are encoded, noise is added. Three inputs are passed to the Discriminator: Correct Text with actual image, Incorrect Text with actual image and Correct Text with fake image. These help in the better training of Discriminator. The dataset used was the Oxford-102 flower dataset.

The authors[8] implemented a StackGAN to generate a stylised output image directly from the model. The Stage 1 GAN generated low resolution images and a new conditioning is added to the Stage 2 GAN to generate higher resolution images in a given style. The discriminator is trained on stylised 256 x 256 images. The datasets used were COCO and CUB. No meaningful results were obtained from this method. The reason given, too much time to reasonably generate enough stylized training data and to perform a hyper-parameter search with enough iterations each time to find the best settings.

This paper[9] proposed TReCS (Tag - Retrieve - Compose - Synthesize) that uses descriptions to retrieve segmentation masks and predict object labels aligned with mouse traces. These alignments are used to select and position masks to generate a fully covered segmentation canvas; the final image is produced by a segmentation-to-image generator using this canvas. The dataset used Localised Narratives, has detailed natural language descriptions of images paired with mouse traces that provide a sparse, fine-grained visual grounding for phrases. In TReCS, a tagger predicts object labels for every word. A BERT model is trained for this. • A text-to-image dual encoder retrieves images with semantically relevant masks. This helps select contextually appropriate masks for each object. Selected masks are composed corresponding to trace order, with separate canvases for background and foreground objects. Finally, a realistic image is synthesized by inputting the complete segmentation to mask-to-image translation models. The evaluation metrics considered were Inception Score and Fréchet Inception Distance. The model is compared with AttnGAN. Both the models are evaluated on the COCO validation set of Localized Narratives (LN-COCO) and on a held out test set of Open Images data that is covered by Localized Narratives (LN-OpenImages). In the case of LN-COCO, TReCS has a better IS and FID, i.e., 21.3 and 48.7, respectively. On the other hand, for LN-OpenImages, AttnGAN has a better IS and FID, i.e., 15.3 and 56.6, respectively.

This study[10] proposed Recurrent Convolutional Generative Adversarial Network (RC-GAN). Conditional GANs were used with recurrent neural networks (RNNs) and convolutional neural networks (CNNs) for generating meaningful images from a textual description. RNN was used for capturing the contextual information of text sequences by defining the relationship between words at altered time stamps. Text-to-image mapping was performed using an RNN and a CNN. The CNN recognized useful characteristics from the images without the need for human intervention. An input sequence was given to the RNN, which converted the textual descriptions into word embeddings with a size of 256. These word embeddings were concatenated with a 512-dimensional noise vector. Semantic information from the textual description was passed into the generator. This generated image was used as input in the discriminator along with real/wrong textual descriptions and real sample images from the dataset. The dataset used was Oxford-102 flowers. The evaluation metrics were Inception Score and PSNR. The PSNR value of the model obtained was 30.12 dB. The model was compared with GAN-INT-CLS, StackGAN, StackGAN++, HDGAN, and DualAttn-GAN for the Inception Score. RC-GAN gave the best score, i.e., 4.15 ± 0.03.

The paper[15] proposed a two-stage model: a prior that generates a CLIP image embedding given a text caption, and a decoder that generates an image conditioned on the image embedding. The decoder used diffusion models to produce images conditioned on CLIP image embeddings. To generate high resolution images, two diffusion upsampler models were trained: one to upsample images from 64×64 to 256×256 resolution, and another to further upsample those to 1024×1024 resolution. For the first upsampling stage, gaussian blur was implemented and for the second, a more diverse BSR degradation, to make the decoder more robust. For the prior, two model classes were explored: Autoregressive and Diffusion. In the Autoregressive (AR)Model, the dimensionality of the CLIP image embeddings is reduced by applying Principal Component Analysis. After applying PCA, the principal components were ordered by decreasing eigenvalue magnitude, quantising the dimensions into discrete buckets and predicting the resulting sequence using a Transformer model with a causal attention mask. The AR prior is conditioned on the text caption and the CLIP text embedding by encoding them as a prefix to the sequence. For the diffusion prior, a decoder-only Transformer was trained with a causal attention mask on a sequence consisting of, in order: the encoded text, the CLIP text embedding, an embedding for the diffusion timestep, the noised CLIP image embedding, and a final embedding whose output from the Transformer is used to predict the un-noised CLIP image embedding. No conditioning was done. The model is not directly trained on the MS-COCO training set, but can still generalize to the validation set zero-shot. With the diffusion prior, the zero shot FID score obtained is 10.39.

The authors[16] introduced a framework called Semantic-Spatial Aware GAN for synthesizing images from input text which consists a simple and effective Semantic-Spatial Aware block, which learns semantic-adaptive transformation conditioned on text to effectively fuse text features and image features and

learns a semantic mask in a weakly-supervised way that depends on the current text-image fusion process in order to guide the transformation spatially. The model had a text encoder that learns text representations, a generator that had 7 SSA blocks for deepening text-image fusion and improving resolution and a discriminator that is used to judge whether the generated image is semantically consistent to the given text. The text encoder was a bidirectional LSTM and pre-trained using real image-text pairs by minimizing the Deep Attentional Multimodal Similarity Model (DAMSM) loss. Each SSA block consisted of an upsample block, a semantic mask predictor, a Semantic-Spatial Condition Batch Normalization block with a residual connection. The upsample block was used to double the resolution of image feature maps by bilinear interpolation operation. The residual connection was used to maintain the main contents of the image features to prevent text-irrelevant parts from being changed and the image information being overwhelmed by the text information. The discriminator concatenated the features extracted from the generated image and the text vector for computing the adversarial loss through two convolution layers. Associated with the Matching-Aware zero-centered Gradient Penalty (MA-GP), it guided the generator to synthesize more realistic images with better text-image semantic consistency. The datasets used were COCO and CUB and the evaluation metrics were Inception Score (IS), Fréchet Inception Distance (FID) and R-precision. The model was compared with StackGAN++, AttnGAN, ControlGAN, SD-GAN, DM-GAN, DF-GAN and DAE-GAN. For the CUB dataset, the model had the best IS, i.e, 5.17±0.08 and for the COCO dataset, the model had the least FID, i.e., 19.37. DAE-GAN had best R-precision in CUB and COCO, 85.4±0.57 and 92.6±0.50, respectively.

The paper[17] introduced MirrorGAN. It consisted of three modules: a semantic text embedding module (STEM), a global-local collaborative attentive module for cascaded image generation (GLAM), and a semantic text regeneration and alignment module (STREAM). In STEM, a RNN is used to extract semantic embeddings from the given text description, which include a word embedding and a sentence embedding. Conditioning Augmentation was done. In GLAM, three image generation networks were stacked sequentially. It took a word embedding and a visual feature as the input. The word embedding was first converted into an underlying common semantic space of visual features by a perception layer. It was multiplied with the visual feature to obtain the attention score. For sentence-level model, the same actions were performed, just like with the word-level model. In STREAM, the text description is regenerated from the generated image, which was semantically aligned with the given text description. The image encoder was a CNN pre-trained on ImageNet and the decoder was a RNN. The datasets used were COCO and CUB and the evaluation metrics were Inception Score (IS) and R-precision. The model was compared with AttnGAN. The IS was best in MirrorGAN for both CUB and COCO, i.e, 4.56 ± 0.05 and 26.47 ± 0.41, respectively.

The authors[18] proposed a way to improve the quality of the images using dialogue. VisDial dialogues were used along with the MS COCO dataset. The model is built upon the StackGAN model. It generates an image in two stages where Stage-I generates a coarse 64 × 64 image and StageII generates a refined 256 × 256 image. Caption embedding was done using a pre-trained encoder. The dialogue embeddings were done by two methods: Non-recurrent encoder and Recurrent encoder. In Non-recurrent encoder, the entire dialogue is collapsed into a single string and encoded with a pre-trained Skip-Thought encoder. In Recurrent encoder, Skip-Thought vectors are generated for each turn of the dialogue and encoded with a bidirectional LSTM-RNN. Conditioning Augmentation is done on the concatenation of the caption and dialogue embeddings, which is passed as the input. In Stage 1, the conditioned variables are concatenated with noise. The generator upsamples this input. In the discriminator, the conditioned variable is concatenated with the downsampled image. This was further downsampled to scalar value between 0 and 1. In Stage 2, the generator, the conditioned variable is concatenated with the downsampled generated image from Stage 1. For Stage-II training, in case of the recurrent dialogue encoder, the RNN weights are copied from Stage-I and kept fixed. The concatenated input is passed through a series of residual blocks and is then upsampled. The Stage-II discriminator, the conditioned variable was concatenated with the downsampled image, which was further downsampled to scalar value between 0 and 1. The evaluation metric was Inception Score (IS). The model was compared with the original StackGAN model. The IS for the recurrent model was 9.74 ± 0.02 and for the non-recurrent model was 9.43 ± 0.04, both the scores higher than the IS of the StackGAN, i.e, 8.45 ± 0.03.

The paper[19] introduced Imagen. It consisted of a text encoder that maps text to a sequence of embeddings and a cascade of conditional diffusion models that map these embeddings to images of increasing resolutions. The model explored three pre-trained text encoders: BERT, T5 and CLIP. The weights were frozen of these text encoders. The model utilized a pipeline of a base 64×64 model and two text-conditional super-resolution diffusion models to upsample a 64 × 64 generated image into a 256 × 256 image and then to 1024 × 1024 image. Noise conditioning augmentation is used for both the super-resolution models. For the base model, the U-Net architecture is adapted. The network is conditioned on text embeddings via a pooled embedding vector, added to the diffusion timestep embedding. Further conditioning on the entire sequence of text embeddings was done by adding cross attention over the text embeddings at multiple resolutions. For the Super-resolution model, modifications were made on the U-net model like removing the self-attention layers. Cross attention is used in the super-resolution models. The input for this was the 64x64 low-resolution images and output was the upsampled 1024 × 1024 images as outputs. The dataset used was COCO and the evaluation metric was Zero-shot FID-30K, in which the model had the distance of 7.27.

Description of the Dataset

The dataset that we used in the project is one of the well-known datasets for Text-to-Image synthesis, i.e., the Oxford 102 Flower dataset. Oxford-102 contains 8,192 images from 102 categories of flowers. Each category has 40-258 images. We imported this from Kaggle[11]. The images have large scale, pose and light variations. In addition, there are categories that have large variations within the category and several very similar categories.

In order to generate images from the text, we need the sample captions/text description associated with all images in the dataset. These captions will then be combined with general embeddings for flowers in the GloVe model. This caption data is hosted under the name “cvpr2016_flowers”. We downloaded the captions from a GitHub repository[13]. For every image in the dataset, there are 5 captions, making a total of 40960 captions. The screenshots shown below show the organization of the raw data.

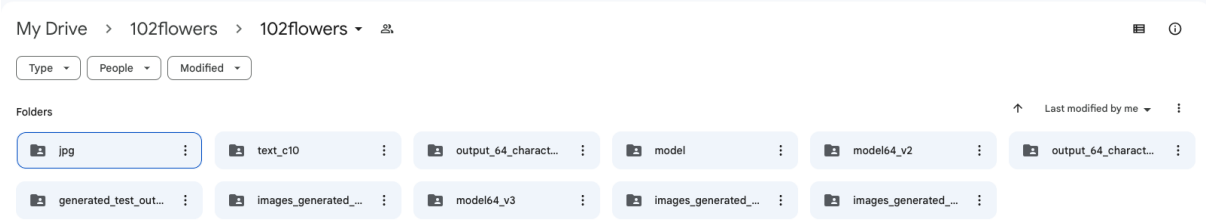


Image 1: Raw data organization

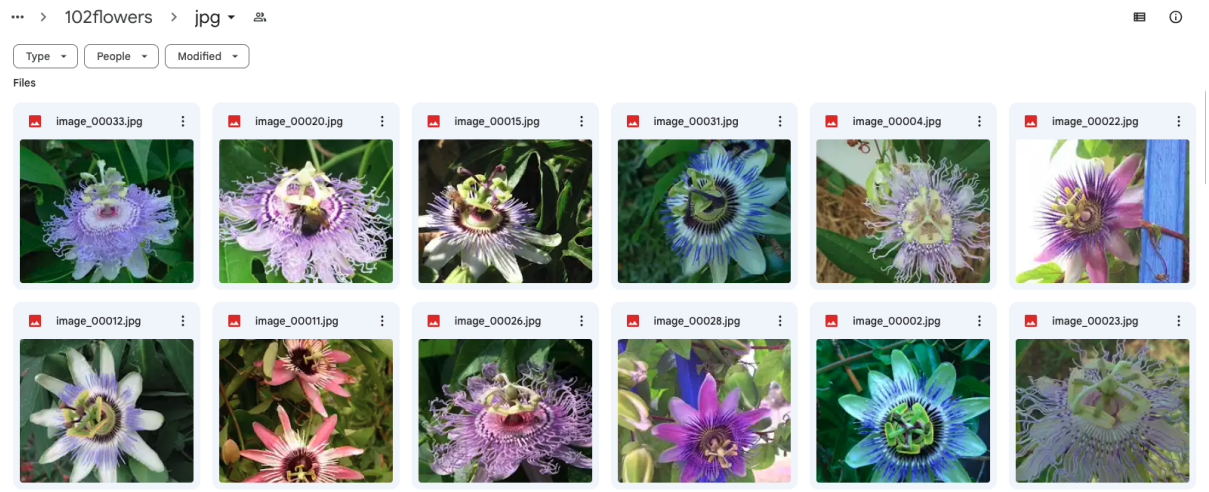


Image 2: Flowers image data in jpg folder

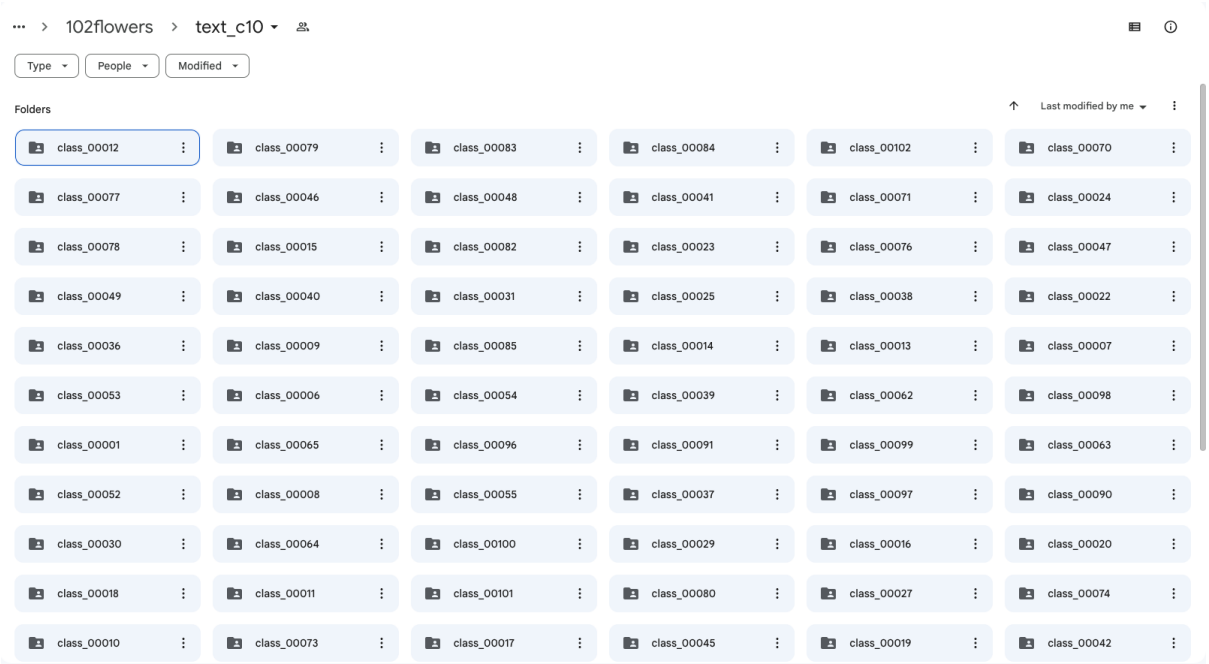


Image 3: Captions data taken from cvpr2016_flowers. There are 102 classes.

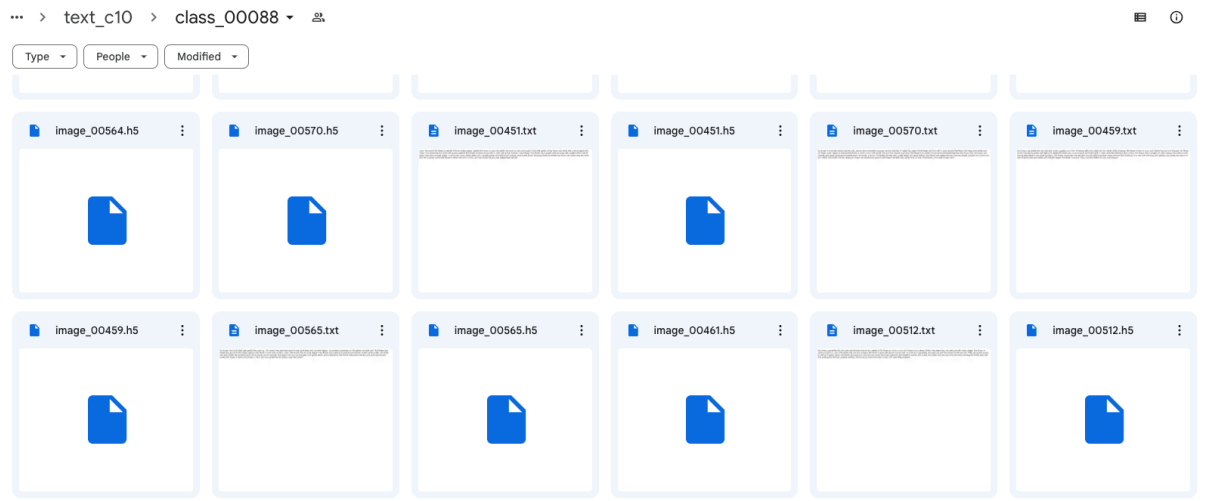


Image 4: Image showing class_00088 folder containing captions in .h5 and .txt format

```
what i like about this flower is candle lit flame shape petals.
pdicels are bown in color,the petals are white in color,and larger in size
the petals of this flower are white with a short stigma
this flower has petals that are white with purple patches
this flower is white and purple in color, with petals that are oval shaped.
this flower has a purple pedicel along with upright white colored petals that have smooth edges.
a flower with many white petals with a purple center and long brown pedicel.
whit & pink flower blooming fromt he center
this flower has petals that are white and has a purple center
this flower is white and pink in color, and has petals that are oval shaped and twisted.
```

Image 5: Sample caption file in .txt format

Methodology

This section provides details above the base model, and the process from data preparation to prediction. In the prediction the images are generated by looking at the prompt.

Base model: DC-GAN

This model uses the deep convolutional generative adversarial networks[5] to generate images from text prompts. We can use this model on different categories of images, like: faces, birds, flowers, animals, etc. We have used it on flower images. The model is capable of learning text representations from words and characters automatically using generative and discriminative capabilities. As a deep convolutional model it has many convolutional layers. The base model architecture is shown in the image below.

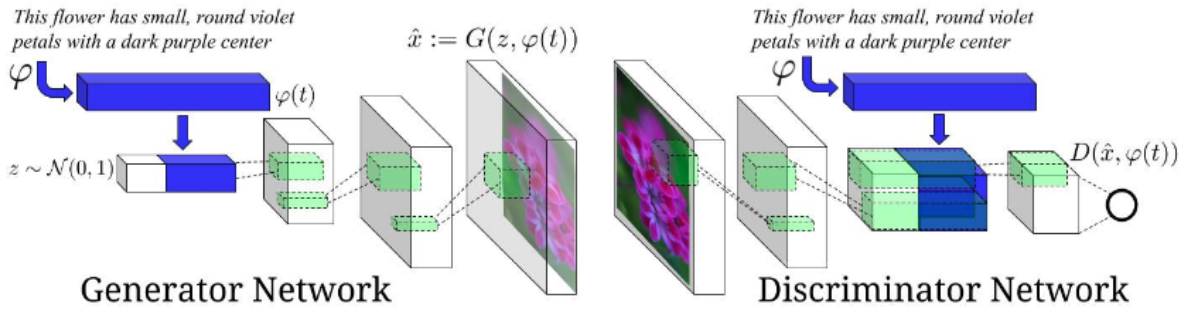


Image 6: GAN-CLS algorithm model architecture[5]

The GAN-CLS algorithm is shown in the image below, which is a matching-aware discriminator. In this approach, the DC-GAN is trained using (text, image) pairs. The discriminator is trained to observe these pairs and judge the pairs as real or fake. The algorithm is shown in the image below.

Algorithm 1 GAN-CLS training algorithm with step size α, using minibatch SGD for simplicity.

Input: minibatch images x, matching text t, mismatching $\hat{t}$, number of training batch steps S
for $n = 1$ **to** S **do**
  $h \leftarrow \varphi(t)$ {Encode matching text description}
  $\hat{h} \leftarrow \varphi(\hat{t})$ {Encode mis-matching text description}
  $z \sim \mathcal{N}(0,1)^Z$ {Draw sample of random noise}
  $\hat{x} \leftarrow G(z, h)$ {Forward through generator}
  $s_r \leftarrow D(x, h)$ {real image, right text}
  $s_w \leftarrow D(x, \hat{h})$ {real image, wrong text}
  $s_f \leftarrow D(\hat{x}, h)$ {fake image, right text}
  $\mathcal{L}_D \leftarrow \log(s_r) + (\log(1 - s_w) + \log(1 - s_f))/2$
  $D \leftarrow D - \alpha \partial \mathcal{L}_D / \partial D$ {Update discriminator}
  $\mathcal{L}_G \leftarrow \log(s_f)$
  $G \leftarrow G - \alpha \partial \mathcal{L}_G / \partial G$ {Update generator}
end for

Image 7: GAN-CLS algorithm[5]

This is a simple yet powerful and accurate way of creating images from text. We have developed our code using this model as our base model.

Importing Libraries and GloVe Embedding

The process started with mounting Google Drive with Colab. We imported a few libraries like NumPy, pandas, etc. In order to make our algorithm understand the captions more meaningfully, we need to create embeddings for the same. The term embeddings, means numeric vector representation of words or phrases. They improve the ability of the computer to understand the semantic meaning words/phrases convey to the model. There are many methods available to create the embeddings, some examples are GloVe embedding, gecko embedding, etc. For our project we used GloVe Embedding. It is an unsupervised learning algorithm for obtaining vector representations for words. In the function, the GloVe word embedding file is opened in read mode using UTF-8 encoding. In the iteration of each line in the GloVe word embedding file, the line is split into a list of tokens, the word and the embedding vector is extracted and the embedding vector is converted to a NumPy array. The GloVe model is loaded using 40000 words.

Paths are set up for the training images and captions. Then, we imported packages required to create a GAN system in Python/Keras. We then set the parameters that helped us feed the images with good resolution. While training the model, we get 28 images to preview. At the end, we get 1 image, when we enter the prompt. There are 5 stages involved in the process of generation of images. We explain them in the sub sections below.

Data Pre-processing

In the data pre-processing step, the images are processed into a binary file. This helped us in quickly accessing, whenever we wanted to use the file. The dimensions of the image are also encoded into the filename of the binary file. Just like the images, the captions are also processed into a binary file. We saved the caption embeddings using NumPy. We used the 64x64, 128x128 pixel image size. The images below show the preprocessed files organization.

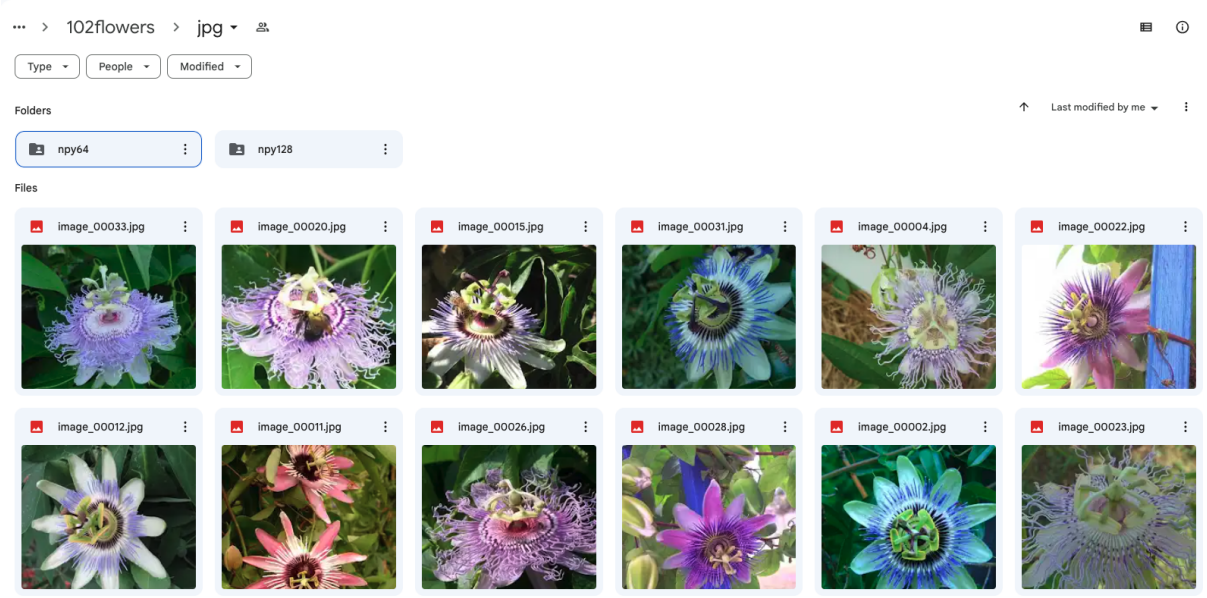


Image 8: folders npy64 and npy128 containing binary files

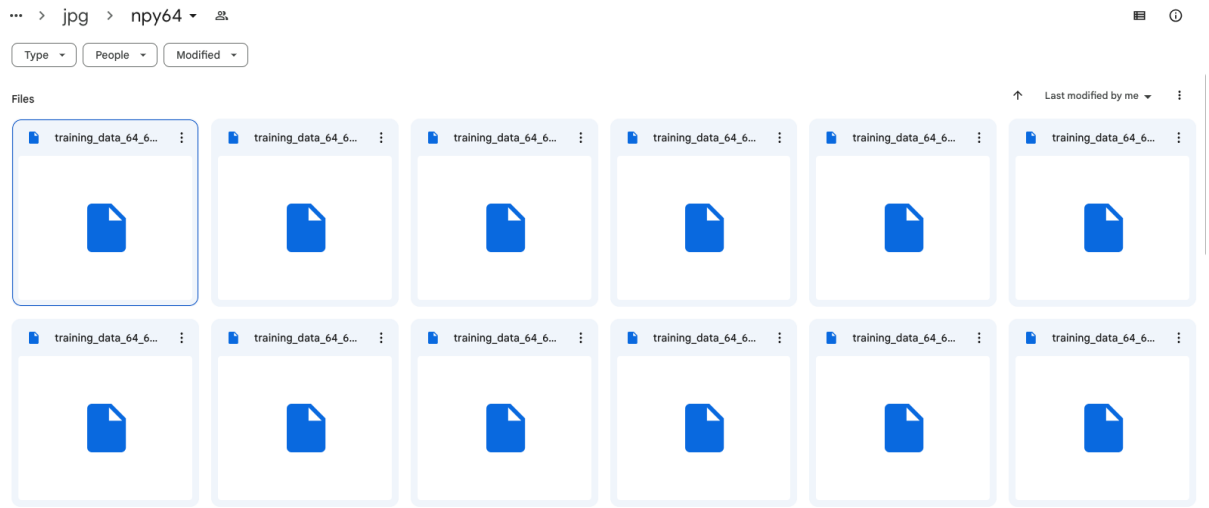


Image 9: expanded view of npy64 folder showing the training data 64x64 binary image files

Loading and Combining NumPy

In this section, we prepared our data splits for training and testing. We created a list to store the preprocessed images and a DataFrame to store the captions. We saved some images , caption embeddings and captions for testing. We, then, shuffled the data for the training dataset. Finally, we created the combined dataset of these images and caption_embeddings.

Data Modelling

In this section, we defined the generator and discriminator. In DC-GAN generator and discriminator functions work in minmax approach just like in gaming. Using these two functions the algorithm further comes to a conclusion about which images are correct/appropriate output. We created a function to save the images into a folder. We initialized the generator and discriminator. To see the output from the generator, we add noise with the caption embeddings. A helper function was defined for computing the loss for discriminator and generator. Adam optimizer is used for both the generator and discriminator.

Training

We defined two functions for the training. The training started with 1 epoch, after which we got the weights file for the generator and discriminator. We used the weights for the generator to check the improvement in the output image from the generator. Gradually, we changed the epochs in the training to 10, 100, 200, 500 and 1000 and obtained the updated weights file for all these epochs. The quality of the generated image improved with the increase in the number of epochs. The images obtained after the training are stored in a folder in the form of a 4 x 7 grid (28 images), in Google Drive. The following image shows that folder with some sample images.

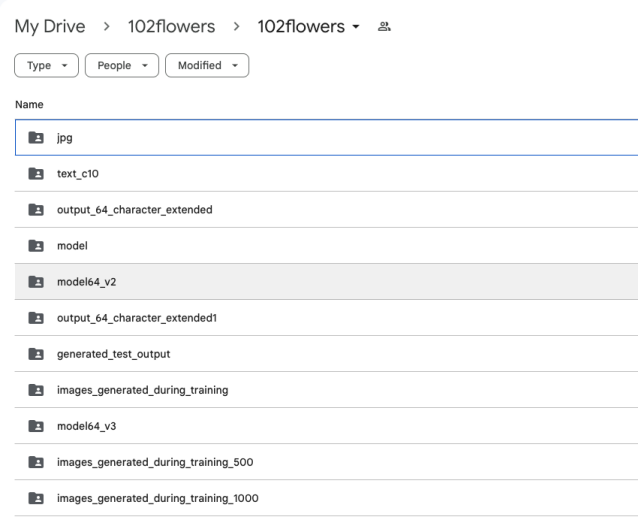


Image 10: Folders "images_generated_during_training_*" contain the images created during training process

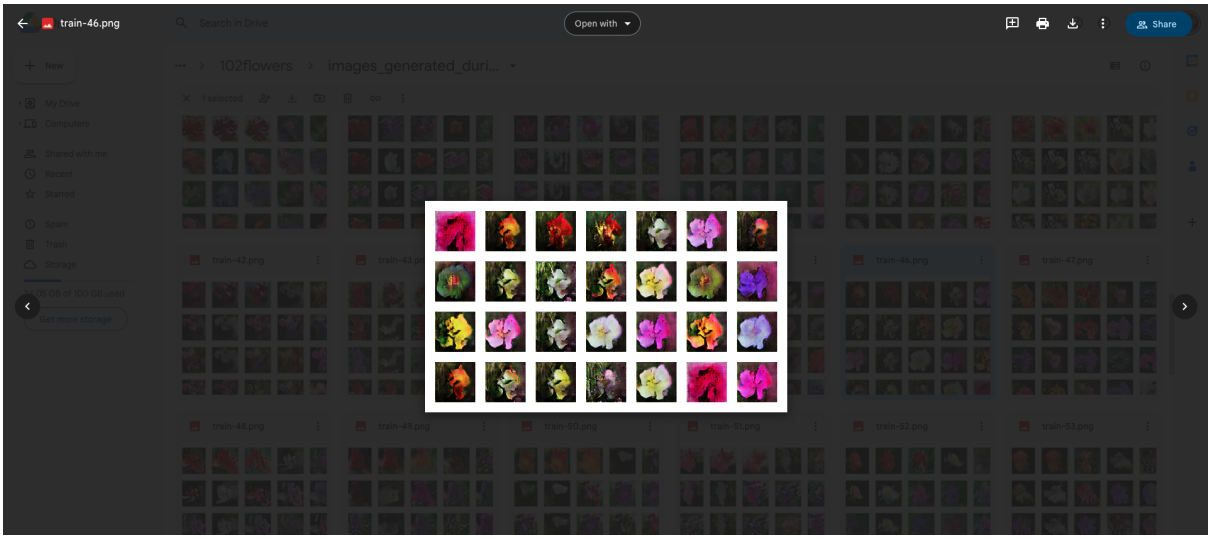


Image 11: .png file containing 28 images from training epoch 46

Generating images using trained model

This section is our testing segment, where using the trained model we generate the images. Two functions were defined:

- one to save the test images generated by the prompt and
- another to display the images.

The image is saved in the folder in Google Drive. The folders named "generated_test_output" contain the .png files, one for each prompt. The following image shows its organization.

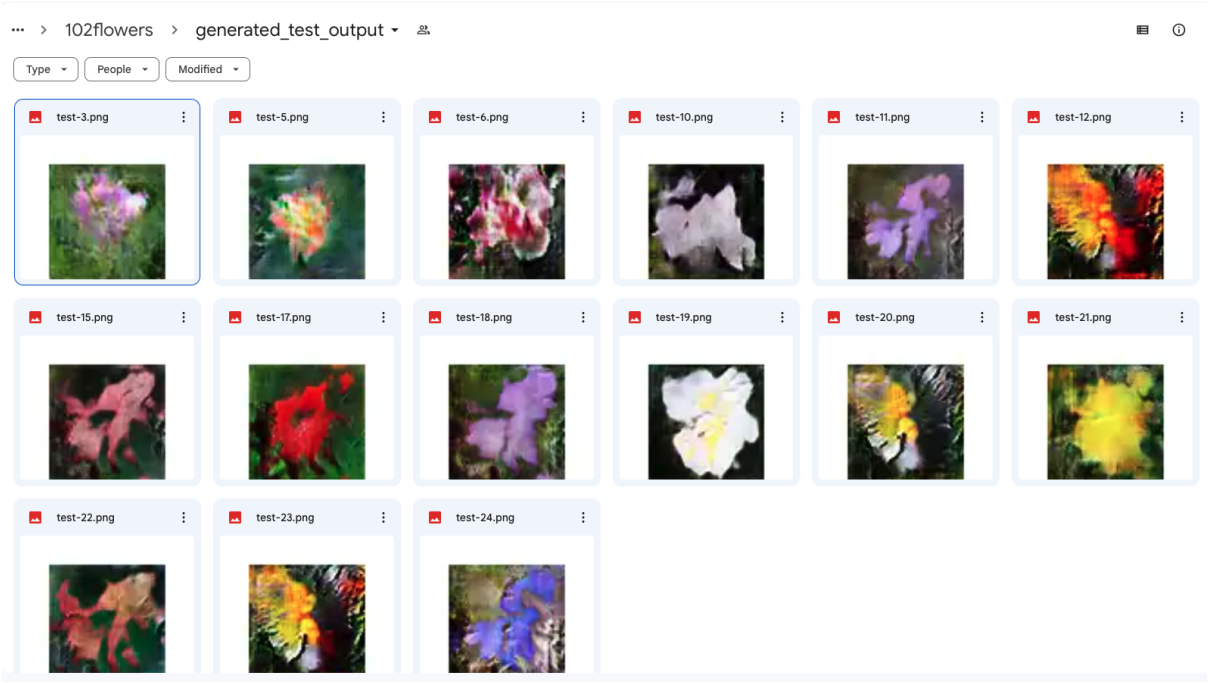


Image 12: Images generated for prompts provided in testing phase

We trained our model for approximately 1800+ epochs and in each epoch nearly 16000+ images were processed with their captions in a randomized way. During the testing, we provided multiple prompts to check the image's appropriateness. We used three types of prompts:

1. Prompts very similar to caption data,
2. Prompts created using some combination of words from the embeddings, and
3. Prompts which are totally random.

The following table shows some prompts and the resulting images for the same.

Table 1: Prompts and images generated during the testing phase

Prompt	Image generated
Prompts very similar to caption data	
the petals on this flower are white with an elaborate pistil.	
a flower that has vivid yellow petals and dark purple stamen	
Prompts created using some combination of words from the embeddings	

this is a blue flower with round petals	
the pistil is white and is very noticeable, and the petals are purple.	
Prompts which are totally random	
rose	
white flower with yellow stigma	

Technology Stack

The code of this project is written in Python programming language. Some of the libraries used**:**

1. **NumPy:** Used for numerical operations, especially for handling arrays and mathematical computations efficiently.
2. **IPython display:** Used for interactive displays
3. **Pandas:** Used for data analysis and associated manipulation of tabular data in DataFrames.
4. **Keras:** Used for implementing the neural networks. Various layers of keras were imported:
    - Input, Reshape, Dropout, Dense, Concatenate
    - Flatten, BatchNormalization
    - Activation, ZeroPadding2D
    - LeakyReLU
    - UpSampling2D, Conv2D, Conv2DTranspose

We used the Python development environment and the code was written in Google Colab. The use of IPython.display is specific to IPython environments and enhances interactive display capabilities. In Google Colab, we executed our code on 3 runtime types, viz., CPU, T4 GPU, and A100 GPU. The time taken for each epoch:

Table 2 - Time taken for 1 epoch in the runtime types

Runtime Type	Time Taken
CPU	40 min
T4 GPU	30 sec
A100 GPU	3 sec

Experimental Results

Coding a GAN for the first time was an exciting experience. There were many situations when the code didn't behave as intended, training took hours to complete, running out of resources in the Colab environment, etc. In spite of these hurdles, we were successful in implementing GAN for the first time. As mentioned before, the algorithm uses a deep convolutional approach to generate the images. This is a very simple and basic implementation technique. We have used 64x64 pixel images hence the generated images are not high resolution.

During the process we observed how the model became more and more accurate in understanding the prompts and generating the images. After 1800+ epochs training we were successful in getting the images generated shown in Table 1. As this is a qualitative analysis, this algorithm uses the generator loss function, discriminator loss function and helper function implementing cross entropy to generate optimal images which do not look fake.

Conclusions

During this study, we were successful in building our first GAN for image generation. As mentioned before, the approach is simple and helps us in understanding the core components of the model. Our model was able to generate 64x64 pixel images as per the text prompt and was able to understand the text prompt fairly well in many cases. This study has motivated us to continue in this domain of generative capabilities in AI models. There were some limitations explored during this study. They are listed below:

- The image resolution has a great impact on the time and resources needed for model training and prediction,
- DC-GAN requires more iterations (thousands of epochs) to start showing permissible results,
- The dataset size matters a lot, if we could bring in a bigger dataset, the model will generate more appropriate images and learns well,
- The execution framework/environment is important as it can make the development process simple and quick.

Future Scope

This study has motivated us positively to explore the domain further and below are some points to taking this study to the next level.

1. Using different GAN implementations to check the accuracy of generated images and comparing the performance,
2. Using different image resolutions and checking its impact on model's learning capabilities,
3. Implementing the algorithms using Tensorflow and advanced frameworks to take advantage of distributed training and prediction,
4. Integrating the trained model in a real application where real time image generation can happen,

5. Implementing a stable diffusion algorithm step by step.

References

1. Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., & Lee, H. (2016, June). Generative adversarial text to image synthesis. In *International conference on machine learning* (pp. 1060-1069). PMLR.
2. Qiao, T., Zhang, J., Xu, D., & Tao, D. (2019). Learn, imagine and create: Text-to-image generation from prior knowledge. *Advances in neural information processing systems*, *32*.
3. Li, B., Qi, X., Lukasiewicz, T., & Torr, P. (2019). Controllable text-to-image generation. *Advances in Neural Information Processing Systems*, *32*.
4. Zhang, H., Goodfellow, I., Metaxas, D., & Odena, A. (2019, May). Self-attention generative adversarial networks. In *International conference on machine learning* (pp. 7354-7363). PMLR.
5. Fu, A., & Hou, Y. (2017). Text-to-image generation using multi-instance stackgan. *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Sprint 2017*, 225-231.
6. Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., ... & Sutskever, I. (2021, July). Zero-shot text-to-image generation. In *International Conference on Machine Learning* (pp. 8821-8831). PMLR.
7. Singh, Akanksha & Anekar, Sonam & Shenoy, Ritika & Patil, Sainath. (2022). Text to Image using Deep Learning. International Journal of Engineering and Technical Research.
8. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/reports/custom/15722067.pdf
9. Koh, J. Y., Baldridge, J., Lee, H., & Yang, Y. (2021). Text-to-image generation grounded by fine-grained user attention. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision* (pp. 237-246).
10. Ramzan, S., Iqbal, M. M., & Kalsum, T. (2022). Text-to-Image Generation Using Deep Learning. *Engineering Proceedings*, *20*(1), 16.
11. https://www.kaggle.com/datasets/nunenuh/pytorch-challange-flower-dataset
12. https://github.com/utsav-195/text-to-image-generator-gan
13. https://github.com/zsdonghao/text-to-image
14. https://nlp.stanford.edu/projects/glove/
15. Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, *1*(2), 3.
16. Liao, W., Hu, K., Yang, M. Y., & Rosenhahn, B. (2022). Text to image generation with semantic-spatial aware gan. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 18187-18196).
17. Qiao, T., Zhang, J., Xu, D., & Tao, D. (2019). Mirrorgan: Learning text-to-image generation by redescription. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 1505-1514).
18. Sharma, S., Suhubdy, D., Michalski, V., Kahou, S. E., & Bengio, Y. (2018). Chatpainter: Improving text to image generation using dialogue. *arXiv preprint arXiv:1802.08216*.
19. Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E. L., ... & Norouzi, M. (2022). Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, *35*, 36479-36494.

Appendix

Code

Text to Image synthesis using DC-GAN

This notebook contains the Python and tensorflow code to generate flower images using the text prompts. We are using Oxford flowers dataset for this activity. The embeddings for text are generated using the glove approach. We are storing the preprocessed data, training data, testing data along with model parameters, model files etc. in Google Drive. The notebook is created in Google colab environment.

We have connected and used different execution/runtime environments with this notebook while the training happened. We found that CPU environment is super slow and TPU environment is the most effecient with respect to time needed for training the model.

Getting Ready

Initializing the Google drive mount point and imports necessary libraries, packages for the execution.

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

```
    Mounted at /content/drive
```

```
[ ] import glob
    import pandas as pd
    import urllib.request
    import imageio
    import os
    import numpy as np

    from urllib.request import urlopen
```

We are using Stanford NLP group's GloVe embeddings approach to create the numerical/vector representation of text. Writing embeddings model is beyond scope of this project as our focus is on text to image synthesis. Therefore we have used the pretrained GloVe model. The embeddings model details can be found at: https://nlp.stanford.edu/projects/glove/

```
def loadGloveModel(gloveFile):
    print("Loading Glove Model")
    f = open(gloveFile,'r',encoding="utf8")
    model = {}
    for line in f:
        try:
            splitLine = line.split()
            word = splitLine[0]
            embedding = np.array([float(val) for val in splitLine[1:]])
            model[word] = embedding
        except:
            print(word)
    print("Done.",len(model)," words loaded!")
    return model
```

```
[ ] #Load the GloVe model using 40000 words
    glove_embeddings = loadGloveModel("/content/drive/My Drive/glove_embeddings/glove.6B.300d.txt")
```

```
    Loading Glove Model
    Done. 400000  words loaded!
```

```
# Setting up the paths
train_data_path = "/content/drive/My Drive/102flowers/102flowers"
train_images_path = "/content/drive/My Drive/102flowers/102flowers/jpg"
train_captions_path = "/content/drive/My Drive/102flowers/102flowers/text_c10/captions"
```

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

     import numpy as np
     from PIL import Image
     from tqdm import tqdm
     import os
     import time
     import matplotlib.pyplot as plt
```

```
[ ]  # code for formatted time string. This will be used to name the files during training, testing, etc.
     def hms_string(sec_elapsed):
         h = int(sec_elapsed / (60 * 60))
         m = int((sec_elapsed % (60 * 60)) / 60)
         s = sec_elapsed % 60
         return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

In order to get good quality of images from the model, we need to feed it images with good resolution. This code section sets the values for the parameters which will be used during the training and testing configurations. Our implementation approach is focussed on pixel to pixel representation of images. All images must be square in shape, so we will specify the scaling accordingly. The resoultion will be applicable to both training and testing.

```
GENERATE_RES = 2 # Generation resolution factor
# (1=32, 2=64, 3=96, 4=128, etc.)
GENERATE_SQUARE = 32 * GENERATE_RES # rows/cols (should be square)
IMAGE_CHANNELS = 3

# Preview image - These parameters will be used while training the model. We are asking our model to generate png images made up of 28 images in each file.
#Hence we use the rows and cols representation.
PREVIEW_ROWS = 4
PREVIEW_COLS = 7
PREVIEW_MARGIN = 16

#test image - These parameters will be used while generation/prediction phase of the model. We are asking our model to generate png images made up of 1 image in each file.
TEST_ROWS=1
TEST_COLS=1
TEST_MARGIN=16

# Size vector to generate images from
SEED_SIZE = 100
EMBEDDING_SIZE = 300

# environment variables providing paths and other paramenters for training and generation
DATA_PATH = train_images_path
MODEL_PATH = "/content/drive/My Drive/102flowers/102flowers/model"
EPOCHS = 50
BATCH_SIZE = 64
BUFFER_SIZE = 4000

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

```
Will generate 64px square images.
```

Data Pre-Processing

In this section we have preprocessed our images data. After pre processing we created a buinary file so that we can reload it quickly as and when needed. There were 16000 images which we uploaded first in the Google drive's data folder and then created a binary file out of it. The dimensions of the image are encoded into the filename of the binary file because we need to regenerate it if these change.

```
training_binary_path = os.path.join("/content/drive/My Drive/102flowers/102flowers/jpg/npy64",
                                    f'training_data_{GENERATE_SQUARE}_{GENERATE_SQUARE}_')

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

training_data = []
flowers_path = sorted(os.listdir(DATA_PATH))

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

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

Captions Pre-processing

In this sestion, we will preprocess the .txt files which contain 5 captions for each image in the dataset. There are 8200+ caption files present. We created embeddings for captions using numpy function. For each 100 files in caption embeddings we added them with glove embedding to create more versatile embeddings representation. For simplicity we preprocess them and create a binary file. We can then load the captions data as and when needed.

```
[ ] text_path = "/content/drive/My Drive/102flowers/102flowers/text_c10/captions"
    text_files = sorted(os.listdir(text_path))
    #print(text_files)
    captions = []
    caption_embeddings = np.zeros((len(text_files),300),dtype=np.float32)
    for filename in range(len(text_files)):
        path = os.path.join(text_path,text_files[filename])
        #print(path)
        f = open(path,'r')
        data = f.read()
        data = data.split("\n")
        f.close()
        for d in range(1):
          x = data[d].lower()
          #x = x.replace(" ","")
          captions.append(x)
          count = 0
          for t in x:
            try:
              caption_embeddings[filename] += glove_embeddings[t]
              count += 1
            except:
              print(t)
              pass
          caption_embeddings[filename] /= count
        if filename %100 == 0:
          print("-------------Files completed:",filename)
```

```
[ ] # saving the caption embeddings numpy
    embedding_binary_path = os.path.join('/content/drive/My Drive/102flowers/102flowers/jpg',
            f'embedding_data.npy')
    print("Saving captions embeddings binary...")
    np.save(embedding_binary_path,caption_embeddings)
```

```
Saving captions embeddings binary...
```

Loading and combining numpys

In this section we will start preparing our data splits for training, testing etc.

```
[ ]  embedding_binary_path = os.path.join('/content/drive/My Drive/102flowers/102flowers/jpg',
             f'embedding_data.npy')
```

```
[ ]  caption_embeddings = np.load(embedding_binary_path)
```

```
[ ]  caption_embeddings.shape
```

```
(8209, 300)
```

```
[ ]  image_binary_path = "/content/drive/My Drive/102flowers/102flowers/jpg/npy64/"
     images = os.listdir(image_binary_path)
```

```
[ ]  images[-1]
```

```
'training_data_64_64_108100.npy'
```

```
[ ]  # Creating a list of all the preprocessed images
     final_images = np.load(image_binary_path + images[0])
     for i in images[1:]:
       print(i)
       try:
         final_images = np.concatenate([final_images,np.load(image_binary_path + i)],axis = 0)
       except:
         pass
```

```
training_data_64_64_100200.npy
training_data_64_64_100300.npy
training_data_64_64_100500.npy
training_data_64_64_100400.npy
training_data_64_64_100600.npy
training_data_64_64_100700.npy
```

```
[ ] # Creating a dataframe to store the captions
    df_captions = pd.DataFrame([])
    df_captions['captions'] = captions[:len(final_images)]
```

```
[ ] df_captions.head()
```

	captions
0	prominent purple stigma,petals are white inc olor
1	this flower is blue and green in color, with p...
2	outer petals are green in color and klarger,in...
3	there are several shapes, sizes, and colors of...
4	the stamen are towering over the stigma which ...

```
[ ] captions[:10]
```

```
['prominent purple stigma,petals are white inc olor',
 'this flower is blue and green in color, with petals that are oval shaped.',
 'outer petals are green in color and klarger,inner petals are needle shaped',
 'there are several shapes, sizes, and colors of petals on this complex flower.',
 'the stamen are towering over the stigma which cannot be seen.',
 'this flower is white and purple in color, with petals that are oval shaped.',
 'the petals of this flower are green with a long stigma',
 'the blossom has a layer of rounded purple and white petals topped by a layer of fringed purple petals.',
 'this flower is purple and yellow in color, with petals that are oval shaped.',
 'the petals on this flower are white with an elaborate pistil.']
```

```
[ ] df_captions.to_csv("/content/drive/My Drive/102flowers/102flowers/text_c10/captions.csv",index=None)
```

```
[ ] final_images.shape
```

```
(8100, 64, 64, 3)
```

```
[ ] df_captions.to_csv("/content/drive/My Drive/102flowers/102flowers/text_c10/captions.csv",index=None)
```

```
[ ] final_images.shape
```

```
(8100, 64, 64, 3)
```

```
[ ] df_captions= pd.read_csv("/content/drive/My Drive/102flowers/102flowers/text_c10/captions.csv")
```

```
df_captions.head()
```

	captions
0	prominent purple stigma,petals are white inc olor
1	this flower is blue and green in color, with p...
2	outer petals are green in color and klarger,in...
3	there are several shapes, sizes, and colors of...
4	the stamen are towering over the stigma which ...

```
[ ] len(captions)
```

```
8209
```

```
[ ] #images for testing
    save_images_captions = captions[:28].copy()
    save_images_embeddings = np.copy(caption_embeddings[:28])
    save_images_npy = np.copy(final_images[:28])
```

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

```
[ ] caption_embeddings.shape
```

```
(8100, 300)
```

```
[ ] #shuffling the data for training data set
    p = np.random.permutation(len(final_images))
```

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

```
[ ] final_images_shuffled.shape
```

```
(8100, 64, 64, 3)
```

```
[ ] final_embeddings_shuffled.shape
```

```
(8100, 300)
```

```
# Batch and shuffle the data
# train_dataset = tf.data.Dataset.from_tensor_slices(training_data) \
#     .shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
train_dataset = tf.data.Dataset.from_tensor_slices({'images': final_images,
                                                    'embeddings': caption_embeddings}).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Data Modeling

In this section we start working on core concepts of DC-GAN. We will define the Generator and Discriminator functions. These functions use multipleactivation functions to bring non linearity in the implementation of model.

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

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

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

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

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

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

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

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

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

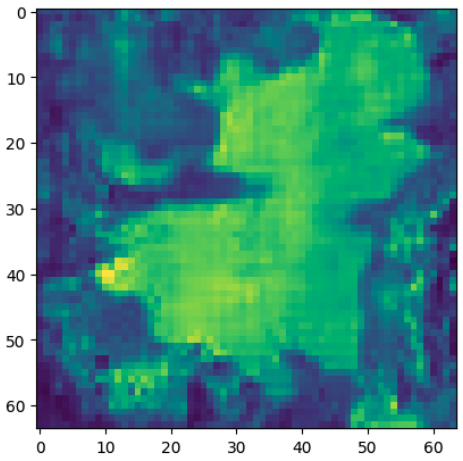
```
def build_discriminator_func(image_shape, embedding_size):
    input_shape = Input(shape=image_shape)
    input_embed = Input(shape=embedding_size)

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

    drop2 = Dropout(0.25)(leaky1)
    conv2d2 = Conv2D(64, kernel_size=4, strides=2, padding="same",kernel_initializer=initializers.RandomNormal(stddev=0.02))(drop2)
    # zero2 = ZeroPadding2D(padding=((0,1),(0,1)))(conv2d2)
    batchNorm2 = BatchNormalization(momentum=0.8)(conv2d2)
    leaky2 = LeakyReLU(alpha=0.2)(batchNorm2)

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

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

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

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

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

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

```
[ ] #Function to save image samples generated
    def save_images(cnt,noise,embeds):
      image_array = np.full((
          PREVIEW_MARGIN + (PREVIEW_ROWS * (GENERATE_SQUARE+PREVIEW_MARGIN)),
          PREVIEW_MARGIN + (PREVIEW_COLS * (GENERATE_SQUARE+PREVIEW_MARGIN)), 3),
          255, dtype=np.uint8)

      generated_images = generator.predict((noise,embeds))

      generated_images = 0.5 * generated_images + 0.5

      image_count = 0
      for row in range(PREVIEW_ROWS):
          for col in range(PREVIEW_COLS):
            r = row * (GENERATE_SQUARE+16) + PREVIEW_MARGIN
            c = col * (GENERATE_SQUARE+16) + PREVIEW_MARGIN
            image_array[r:r+GENERATE_SQUARE,c:c+GENERATE_SQUARE] \
                = generated_images[image_count] * 255
            image_count += 1


      output_path = "/content/drive/My Drive/102flowers/102flowers/images_generated_during_training_1000"
      if not os.path.exists(output_path):
        os.makedirs(output_path)

      filename = os.path.join(output_path,f"train-{cnt}.png")
      im = Image.fromarray(image_array)
      im.save(filename)
```

```
    # Initlializing a generator
    generator = build_generator_func(SEED_SIZE,EMBEDDING_SIZE, IMAGE_CHANNELS)
    generator.load_weights("/content/drive/My Drive/102flowers/102flowers/model/text_to_image_generator_cub_character.h5")
```

```
[ ] # Sample output from the Generator
    noise = tf.random.normal([1, 100])
    generated_image = generator((noise,caption_embeddings[5].reshape(1,300)), training=False)

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

```
<matplotlib.image.AxesImage at 0x7afe1c1878e0>
```




```
    # Initlializing a discriminator
    image_shape = (GENERATE_SQUARE,GENERATE_SQUARE,IMAGE_CHANNELS)

    discriminator = build_discriminator_func(image_shape,EMBEDDING_SIZE)
    discriminator.load_weights("/content/drive/My Drive/102flowers/102flowers/model/text_to_image_disc_cub_character.h5")
```

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

```
    tf.Tensor([[0.31109124]], shape=(1, 1), dtype=float32)
```

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

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

    total_loss = real_loss + fake_loss
    return total_loss

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

Both the generator and discriminator use Adam and the same learning rate and momentum. This does not need to be the case. If you use a **GENERATE_RES** greater than 3 you may need to tune these learning rates, as well as other training and hyperparameters.

```
[ ] # lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    #     initial_learning_rate=2e-4,
    #     decay_steps=100,
    #     decay_rate=0.5)
    generator_optimizer = tf.keras.optimizers.Adam(learning_rate=2.0e-4,beta_1 = 0.5)
    discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=2.0e-4,beta_1 = 0.5)
```

Training the model

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images,captions,fake_captions):
  seed = tf.random.normal([BATCH_SIZE, SEED_SIZE],dtype=tf.float32)

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

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

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

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

```python
# Function that executes trianing process
def train(train_dataset, epochs):
  fixed_seed = np.random.normal(0, 1, (PREVIEW_ROWS * PREVIEW_COLS,
                                       SEED_SIZE))
  fixed_embed = save_images_embeddings

  start = time.time()

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

    gen_loss_list = []
    disc_loss_list = []

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

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

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

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

    generator.save(os.path.join(MODEL_PATH,"text_to_image_generator_cub_character.h5"))
    discriminator.save(os.path.join(MODEL_PATH,"text_to_image_disc_cub_character.h5"))
    print("model saved")
```

```python
train(list(train_dataset.as_numpy_iterator()), 1000)
```

```
epoch start...
now
Epoch 1, gen loss=1.417976975440979,disc loss=1.1559722423553467, 0:00:10.28
1/1 [==============================] - 0s 202ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
model saved
epoch start...
now
Epoch 2, gen loss=1.3062129020690918,disc loss=1.1561769247055054, 0:00:03.11
1/1 [==============================] - 0s 21ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
model saved
epoch start...
now
Epoch 3, gen loss=1.2776546478271484,disc loss=1.1426267623901367, 0:00:03.09
1/1 [==============================] - 0s 22ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
model saved
epoch start...
now
Epoch 4, gen loss=1.3563096523284912,disc loss=1.1284594535827637, 0:00:03.10
1/1 [==============================] - 0s 20ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
model saved
epoch start...
now
Epoch 5, gen loss=1.2954047918319702,disc loss=1.1372756958007812, 0:00:03.07
1/1 [==============================] - 0s 20ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
model saved
```

Generation of Images

```
#Retriving embeddings for testing the results
save_images_embeddings.shape
```

```
(28, 300)
```

```
[36] # Printing the captions
save_images_captions
```

```
['prominent purple stigma,petals are white inc olor',
 'this flower is blue and green in color, with petals that are oval shaped.',
 'outer petals are green in color and klarger,inner petals are needle shaped',
 'there are several shapes, sizes, and colors of petals on this complex flower.',
 'the stamen are towering over the stigma which cannot be seen.',
 'this flower is white and purple in color, with petals that are oval shaped.',
 'the petals of this flower are green with a long stigma',
 'the blossom has a layer of rounded purple and white petals topped by a layer of fringed purple petals.',
 'this flower is purple and yellow in color, with petals that are oval shaped.',
 'the petals on this flower are white with an elaborate pistil.',
 'this flower has petals that are pink with stringy white stamen',
 'this flower has star shaped white petals as its main feature.',
 'lower petals are white in color,and larger in size,inner petals are purple in clor',
 'this flower has petals in the shape of a circle and are purple and stringy',
 'a unique looking flower that has multiple colors and long petals',
 'this flower is white and purple in color, with petals that are oval shaped.',
 'this flower is white in color, with petals that are oval shaped.',
 'this flower has thick star shaped maroon petals as its main feature.',
 'this flower has petals that are white with purple stringy stamen',
 'the pistil is white and is very noticeable, and the petals are purple.',
 'purple flower with curvy string-like petals and a group of large yellow stamen in the center.',
 'this flower has an elaborate golden stamen and two different types of purple petals.',
 'flower with white long white petals and very long purple stamen',
 'this flower is blue and white in color, with petals that are oval shaped.',
 'leaves are green in color,outer petals are white green in color',
 'the petals of the flower are pink in color and have filaments that are pink in color.',
 'this is a purple flower with long purple anthers on it.',
 'this is a strange flower with purple petals and a white stigma.']
```

```python
#Function to save generated image after we provide the text
def save_test_images(cnt,noise,embeds):
  image_array = np.full((
      TEST_MARGIN + (TEST_ROWS * (GENERATE_SQUARE+TEST_MARGIN)),
      TEST_MARGIN + (TEST_COLS * (GENERATE_SQUARE+TEST_MARGIN)), 3),
      255, dtype=np.uint8)

  generated_test_images = generator.predict((noise,embeds))

  generated_test_images = 0.5 * generated_test_images + 0.5

  image_count = 0
  for row in range(TEST_ROWS):
      for col in range(TEST_COLS):
        r = row * (GENERATE_SQUARE+16) + TEST_MARGIN
        c = col * (GENERATE_SQUARE+16) + TEST_MARGIN
        image_array[r:r+GENERATE_SQUARE,c:c+GENERATE_SQUARE] \
            = generated_test_images[image_count] * 255
        image_count += 1


  output_path = "/content/drive/My Drive/102flowers/102flowers/generated_test_output"
  if not os.path.exists(output_path):
    os.makedirs(output_path)

  filename = os.path.join(output_path,f"test-{cnt}.png")
  im = Image.fromarray(image_array)
  im.save(filename)
```

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

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

```
test_image("the pistil is white and is very noticeable, and the petals are purple.",18)
```

```
1/1 [==============================] - 0s 442ms/step
```

```
[51] import IPython
     IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-18.png')
```

```
[52] test_image("the petals on this flower are white with an elaborate pistil.",19)
```

```
1/1 [==============================] - 0s 33ms/step
```

```
import IPython
IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-19.png')
```

```
test_image("This is a red flower.",20)
```

```
1/1 [==============================] - 0s 129ms/step
```

```
[57] import IPython
     IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-20.png')
```

```
[76] test_image("a flower that has vivid yellow petals and dark purple stamen",21)
```

```
1/1 [==============================] - 0s 26ms/step
```

```
[77] import IPython
     IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-21.png')
```

```
test_image("this is a red flower that has vivid petals and light pink stamen",22)
```

```
[79] import IPython
     IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-22.png')
```

```
test_image("this is an orange rose flower",23)
```

```
1/1 [==============================] - 0s 40ms/step
```

```
[81] import IPython
     IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-23.png')
```

```
[84] test_image("this is a blue flower with round petals",24)
```

```
1/1 [==============================] - 0s 32ms/step
```

```
import IPython
IPython.display.Image('/content/drive/My Drive/102flowers/102flowers/generated_test_output/test-24.png')
```