

Text to Image Synthesis Using Text Conditioned Auxiliary Classifier and Generative Adversarial Network

By

Lucky Mathias Kispotta	501
Barnik Roy	505
Sayandip Srimani	537

Under The Guidance of

Dr. Sonali Sen

Submitted to the Department of Computer Science,
ST. XAVIER'S COLLEGE (AUTONOMOUS), KOLKATA.

**St. Xavier's College
(Autonomous), Kolkata**

CERTIFICATE OF AUTHENTICATED WORK

This is to certify that the project report entitled “**Text to Image Synthesis Using Generative Adversarial Networks and Generative Adversarial Network**” submitted to Department of Computer Science, ST. XAVIER'S COLLEGE (AUTONOMOUS), KOLKATA, in partial fulfilment of the requirement for the award of the degree of Bachelor of Science (B.Sc.) is an authentic work carried out by:

Name	Roll No.	Registration No.
Lucky Mathias Kispotta	3-15-20-0501	A01-1132-0886-20
Barnik Roy	3-15-20-0505	A01-1112-0890-20
Sayandip Srimani	3-15-20-0537	A01-1112-0914-20

under my guidance. The matter embodied in this project is authentic and is genuine work done by the student and has not been submitted whether to this College or to any other Institute for the fulfilment of the requirement of any course of study.

Sonali Sen

.....
Signature of the Supervisor

Dr. Sonali Sen

.....
Name of the Supervisor

Date:

5/5/23

Assistant Professor
Department of Computer Science
St. Xavier's College [Autonomous]
30 Mother Teresa Sarani
Kolkata – 700016

ABSTRACT

The project aims to generate realistic images from textual descriptions using Text Conditioned Auxiliary Classifier Generative Adversarial Network. TacGAN is a generative adversarial network (GAN) that uses a text encoder to encode the textual description and a conditional generator to generate the corresponding image. The generator is trained to minimize the difference between the generated image and the real image, while the discriminator is trained to distinguish between real and generated images.

Generative Adversarial Networks (GANs) have found some very interesting implementations in the past year like a deepfake that can animate your face with just your voice, a neural GAN to fight fake news, a CycleGAN to visualize the effects of climate change, and more. Text-to-image synthesis using TAC-GAN is also a challenging task that involves developing an effective way to encode text into a visual feature space, using attention mechanisms to focus on relevant parts of the text, and training the GAN to generate high-quality images that are faithful to the input descriptions.

The developed methods have shown prominent progress on visual quality of the synthesized images, but it still faces challenges in the image synthesis of details. The objective of doing a project on text to image synthesis using TAC-GAN is to generate high-quality images from textual descriptions for various applications such as photo editing or computer-aided content creation. Solving these challenges requires creativity, critical thinking, and problem-solving skills, making it a challenging and rewarding project to work on. In this project there is number of applications of distinctive published papers of many researches encircling GAN architecture.

ACKNOWLEDGEMENT

We wish to express our profound sense of gratitude for our project supervisor Dr. Sonali Sen of St. Xavier's College (Autonomous), Kolkata, for her support inspiration and guidance. She has showed us different ways to approach a problem. We have also learned from her that the goal has to be clear before presenting the approach. We are immensely grateful to her for giving her valuable time and constant advice for discussing ideas related to the project work. It is being a precious learning experience for us to work under her guidance.

We are also thankful to our HOD, respected Professors and staff members of the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata for providing us with the resources for working on the project.

At last, we would like to express gratitude to our friends and our parents and to all who have directly or indirectly extended their valuable guidance and advice during the preparation of this project.

Name and Signature of the Project Team members:



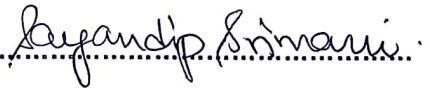
	Name	Signature
1.	Lucky Mathias Kispotta	
2.	Barnik Roy	
3.	Sayandip Srimani	

TABLE OF CONTENTS

1.INTRODUCTION	1
1.1 Background	1
1.2 Objectives	2
1.3 Purpose, Scope and Application	3
1.3.1 Purpose	3
1.3.2 Scope	3
1.3.3 Application	3
2. RELATED WORK AND TECHNOLOGIES	5
3. REQUIREMENTS ANALYSIS	7
3.1 Problem Definition	7
3.2 Requirement Specifications	9
3.3 Software and Hardware Requirements	9
3.3.1 Hardware requirements	9
3.3.2 Software requirements	11
4. SYSTEM DESIGN AND ANALYSIS	13
4.1 Problem Statement AND Conceptual Model	13
4.2 GAN Model OF The Project	15
4.3 Text Conditioned Auxiliary Classifier Generative Adversarial Network (TAC-GAN)	17
4.4 Modules	18
4.4.1 Skip through vector (for data pre-processing)	18
4.4.2 Generative adversarial networks	19
4.4.3 GENERATOR	19
4.4.4 DISCRIMINATOR	20
4.5 IMPLEMENTATION DETAILS	21
4.5.1 GAN training algorithm	22
4.5.2 IMPLEMENTATION	22
4.5.3 TRAINING	23
4.5.4 Loss function	24
4.5.5 Dataset	25
4.6 User Interface	25

5. IMPLEMENTATION AND TESTING	29
5.1 STEP 1: Download/manage data	30
5.2 STEP 2: Data Pre-processing	30
5.3 STEP 3: Describing the GAN Model	36
5.4 STEP 4: TRAINING	46
5.5 STEP 5: GENERATING IMAGE	58
5.6 STEP 6: WEBAPP	63
5.7 TESTING	77
6. RESULTS AND DISCUSSION	81
7. CONCLUTION	84
7.1 Limitations Faced	85
7.2 Future Scope of The Project	85
REFERENCES	88

TABLE OF FIGURES

Fig 1: System flowchart of the problem statement	14
Fig 2: The general semantics flow chart of generative adversarial network (GAN)	15
Fig 3: Samples generated by Generator is termed as fake sample	16
Fig 4: Block diagram of the Generator and Discriminator	16
Fig 5: Skip thought vector process diagram	18
Fig 6: Diagram describing working of the generator	20
Fig 7: Diagram describing working of the discriminator	21
Fig 8: Architecture of GAN	21
Fig 9: Flow diagram of implementation	23
Fig 10: Loss Function	25
Fig 11: Frontend User Interface	27
Fig 12: Display screen of the generated images	28
Fig 13: Information display for user	28
Fig 14: Following is the architecture of our GAN model	29
Fig 15: test result 1	81
Fig 16: test result 2	81
Fig 17: test result 3	82
Fig 18: test result 4	82
Fig 19: test result 5	83

1. INTRODUCTION

1.1 Background

Humans possess a natural ability to create mental images in response to stories they hear or read, which is integral to various cognitive processes such as memory, spatial navigation, and reasoning. Visual mental imagery or “seeing with the mind’s eye” also plays an important role in many cognitive processes such as memory, spatial navigation, and reasoning. Developing a system that can replicate this ability to understand the connection between language and visuals and generate images based on textual descriptions would be a significant advancement towards achieving human-like intelligence. The advent of Generative Adversarial Networks (GANs) made it possible to train generative models for images in a completely unsupervised manner. GANs have sparked a lot of interest and advanced research efforts in synthesizing images.

While Generative Adversarial Networks (GANs) are commonly used for text-to-image synthesis, there have also been other methods that have been explored. Here are some contexts where text-to-image synthesis projects were tried without using GANs:

Variational Autoencoder (VAE): VAE is a generative model that learns a low-dimensional representation of data and can be used for image synthesis. Researchers have explored the use of VAEs for text-to-image synthesis, where a VAE is trained to generate an image from a textual description.

Deep Convolutional Networks (DCN): DCNs are a class of neural networks commonly used for image processing tasks. Researchers have explored the use of DCNs for text-to-image synthesis, where a DCN is trained to generate an image from a textual description.

Neural Style Transfer: Neural Style Transfer is a technique that involves transferring the style of one image to another. Researchers have explored the use of Neural Style Transfer for text-to-image synthesis, where the style of an image is transferred to an image generated from a textual description.

Conditional Variational Autoencoder (CVAE): CVAE is an extension of VAE that allows for the generation of data conditioned on a given input. Researchers have explored the use of CVAEs for text-to-image synthesis, where a CVAE is trained to generate an image conditioned on a textual description.

Adversarial Latent Semantic Analysis (ALSA): ALSA is a method that combines Latent Semantic Analysis (LSA) with adversarial training. Researchers have explored the use of ALSA for text-to-image synthesis, where ALSA is trained to generate an image from a textual description.

Overall, there are several different methods that have been explored for text-to-image synthesis besides GANs, and each method has its own strengths and weaknesses depending on the specific application.

1.2 Objectives

We focus on text-to-image synthesis, which aims to produce an image that correctly reflects the meaning of a textual description. Although the methods presented in this review can be applied to many image domains, most research focuses on methods generating visually realistic, photographic, natural images.

The project aims to develop a system using Generative Adversarial Networks (GANs) to generate realistic images from textual descriptions, thereby bridging the gap between vision and language and advancing the field of artificial intelligence. Overall, the objective of this project on text to image synthesis using Text-Conditioned Auxiliary Classifier-GAN is to contribute to the development of the technology that is revolutionizing a wide range of industries and having a significant impact on society.

Why this area or technology as the project topic is chosen?

GANs are one of the most popular deep learning models used in computer vision and image processing applications. TAC-GAN (Text-Conditioned Auxiliary Classifier Generative Adversarial Network) is a type of GAN that can generate images from text descriptions. This project on text to image synthesis using TAC-GAN technology involves several challenges, including developing an effective way to encode text into a visual feature space, using attention mechanisms to focus on relevant parts of the text, and training the GAN to generate high-quality images that are faithful to the input descriptions. Solving these challenges requires creativity, critical thinking, and problem-solving skills, making it a challenging and rewarding project to work on.

1.3 Purpose, Scope AND Applicability

1.3.1 Purpose:

The text-to-image using GAN project has significant potential to improve the field of artificial intelligence by bridging the gap between language and vision. By training a GAN on textual descriptions and corresponding images, the system can generate new images that accurately reflect the meaning of the given text, allowing for a more seamless integration of language and visual information.

Overall, the text-to-image using GAN project has the potential to revolutionize how we interact with and utilize visual information, from enhancing virtual reality experiences to improving image search and retrieval systems.

1.3.2 Scope:

The scope of the text-to-image using GAN project is to develop a system that can generate realistic images based on textual descriptions. This system has applications in a variety of fields, including virtual reality, gaming, image search and retrieval, and creative industries such as graphic design and advertising. The project will involve training a GAN model on a large dataset of textual descriptions and corresponding images to create a generator that can accurately produce images based on given textual inputs. The scope also includes evaluating the performance of the model using metrics such as image quality, diversity, and similarity to the original textual descriptions. Ultimately, the goal of the project is to develop a system that can generate high-quality images that accurately reflect the intended meaning of the given textual descriptions, which would have significant implications for the field of artificial intelligence and image processing.

1.3.3 Applicability:

The text-to-image using GAN project has broad applicability in various fields where there is a need to generate images based on textual descriptions. Some potential applications of the project include:

Virtual Reality and Gaming: The system can be used to generate realistic images in virtual reality or gaming environments based on textual descriptions, allowing for more immersive experiences.

Image Search and Retrieval: The system can be used to improve image search and retrieval systems by enabling users to search for images using textual descriptions.

Creative Industries: The system can be used in graphic design and advertising to quickly generate images based on textual descriptions, potentially saving time and costs associated with manual image creation.

Accessibility: The system can be used to generate visual descriptions of text for individuals who are visually impaired, allowing them to more easily comprehend the content.

Content Generation: The system can be used to automatically generate images for use in social media posts, news articles, and other forms of online content.

2. RELATED WORK AND TECHNOLOGIES

A large number of paintings on image modelling targeted on other obligations together with photo inpainting or texture technology. These tasks produced new photographs primarily based on a present image. but picture synthesis not counting on such existing records has no longer had a good deal fulfillment until recently in the previous few years, tactics primarily based on Deep studying have raised, which can be able to synthesize pics with numerous fulfillments. Variational Autoencoders (VAE) as soon as skilled, can be interpreted as generative fashions that produce samples from a distribution that approximates that of the training set. The DRAW version extends the VAE architecture by means of the use of recurrent networks and an attention mechanism, remodeling it into a series encoding/deciphering procedure such that “the network decides at each time-step ‘where to examine’ and ‘wherein to jot down’ as well as ‘what to write down’”, and being able to produce tremendously realistic handwritten digits. Mansimov *et al*, extended the DRAW model via using a Bidirectional RNN and conditioning the generating manner on text captions, generating consequences that had been slightly higher than the ones of other countries of the artwork models.

Alternatively, Generative opposed Networks (GAN) have received a considerable amount of hobby considering that its inception, having been utilized in obligations together with unmarried-image awesome resolution, Simulated + Unsupervised studying, photo-to-picture translation and semantic picture inpainting. The basic framework, but has a tendency to produce blurry snap shots, and the excellent of the generated images has a tendency to decrease because the resolution increases. Some procedures to tackle these problems have targeted on iteratively refining the generated photos. For example, within the style and shape GAN (S2-GAN) model, a primary GAN is used to supply the photograph shape, that's then fed right into a 2nd GAN, responsible for the picture fashion. Similarly, the Laplacian GANs (LAPGAN) can have an indefinite quantity of levels, integrating “a conditional shape of GAN model into the framework of a Laplacian pyramid”.

Different tries to solve the troubles of the basic framework have tried making the community higher privy to the facts distribution that the GAN is supposed to model. Through conditioning the input on specific class labels, the Conditional GAN (CGAN) turned into capable of producing higher decision snap shots. instead, the Auxiliary Classifier GAN (ACGAN) has been proven to be able to synthesizing structurally coherent 128×128 pictures through

schooling the discriminator to also classify its input of unique relevance to this work is the conditioning of the generative system additionally on textual content. Reed following the work at the Generative opposed What wherein Networks, had been able to make the synthesized photos correspond to a textual description used as enter, with a decision of sixty-four \times sixty-four. With a comparable approach, the StackGAN version leverages the advantages of the usage of a couple of tiers and is able to exhibit rather practical 256 \times 256 photographs.

3. REQUIREMENTS ANALYSIS

3.1 Problem Definition

Generating images from natural language is one of the primary applications of recent conditional generative models. Besides testing our ability to model conditional, highly dimensional distributions, text to image synthesis has many exciting and practical applications such as photo editing or computer-aided content creation. Nevertheless, these problems are entirely different because text-image or image-text conversions are highly multimodal problems. If one tries to translate a simple sentence such as “This is a beautiful red flower” to French, then there are not many sentences which could be valid translations. If one tries to produce a mental image of this description, there is a large number of possible images which would match this description. Though this multimodal behavior is also present in image captioning problems, there the problem is made easier by the fact that language is mostly sequential. This structure is exploited by conditioning the generation of new words on the previous (already generated) words. Because of this, text to image synthesis is a harder problem than image captioning. paraphrase this without plagiarism. Recent progress has been made using Generative Adversarial Networks (GANs). The problem that we are working on in the project text to image using GAN is to develop a system that can generate high-quality images from textual descriptions. This problem falls under the broader domain of computer vision and natural language processing.

To achieve this goal, we can divide the problem into several sub-problems:

1. Text encoding: The first sub-problem is to encode the textual descriptions into a format that can be used by the GAN to generate images. This involves using natural language processing techniques such as tokenization and embedding to convert the text into a numerical format.
2. GAN architecture: The second sub-problem is to design an appropriate GAN architecture that can generate high-quality images from the encoded text. This involves choosing appropriate neural network architectures for the generator and discriminator, as well as deciding on the loss functions and training algorithms to use.

3. Training data: The third sub-problem is to collect and preprocess a large dataset of text-image pairs that can be used to train the GAN. This involves identifying appropriate sources of data, cleaning and annotating the data, and splitting the data into training, validation, and testing sets.
4. Extract the skip-thought features for the captions and prepare the dataset: For embedding the textual descriptions of the images into vectors we used skip-thought vectors. Data preprocessing involves several steps to prepare the text and image data for training the model. Firstly, the text data is tokenized and encoded into a fixed-length sequence of integers using a word embedding technique such as GloVe or Word2Vec. The image data is resized and normalized to a fixed size and pixel range. Next, the text and image data are paired up to form a dataset of text-image pairs. The text and image data may come from different sources, so it is important to ensure that the pairs are correctly matched and aligned. Finally, the preprocessed data is fed into the GAN model for training, with appropriate hyperparameters and loss functions selected to optimize the model's performance.
5. Training process: The fourth sub-problem is to train the GAN on the text-image dataset. This involves setting up the training pipeline, choosing appropriate hyperparameters, and monitoring the training process to ensure that the GAN is learning effectively.
6. Image quality evaluation: The fifth sub-problem is to evaluate the quality of the generated images. This involves using appropriate metrics such as Inception Score and Frechet Inception Distance to compare the generated images to the ground truth images, and iteratively refining the GAN architecture and training process to improve image quality.

Overall, the problem of text to image synthesis using GAN can be divided into these sub-problems, each of which requires careful consideration and implementation to successfully generate high-quality images from textual descriptions.

3.2 Requirements Specifications

Constraints:

- Time
- Computing Resources

Time will be a constraint in terms of development of the GAN architecture and training the GAN. The time required for training a GAN model for text to image synthesis can vary depending on various factors such as the size and complexity of the dataset, the hardware used for training, and the specific hyperparameters chosen for the model. In general, training a GAN model can take a considerable amount of time, often ranging from several days to a few weeks or even longer. For instance, a GAN model trained on the COCO dataset, which contains around 330k images, may take several weeks or if trained on the Oxford 102 Flower dataset, which contains approx. 8k images may take 1 to 2 weeks, to train on a single GPU.

In general, training a GAN model requires a significant number of computational resources, particularly in terms of GPU memory and processing power. For example, training a GAN model on a dataset like COCO, which contains around 330k images, may require a GPU with at least 11 GB of memory or more, depending on the batch size used for training. The experiments are generally carried out on two Intel Xeon machines with GPUs. One of the machines has a Nvidia GeForce GTX 750 Ti and 8 GB RAM while the other has Nvidia GeForce GTX 1080 Ti (12GB). The runtime is usually observed to be approximately 24 hours for majority of the trials carried.

3.3 Software AND Hardware Requirements

3.3.1 Hardware requirements:

The hardware requirements for preprocessing and training a GAN model for text to image synthesis can vary depending on the size of the dataset, the complexity of the model, and the desired training time.

For preprocessing to acquire skip thought vectors of text captions of image datasets, the following hardware requirements may be needed:

- A computer with at least 8GB of RAM and a modern CPU (e.g., Intel i5 or i7) for processing the textual data and preparing it for training.
- Sufficient storage space for storing the preprocessed data, which can range from a few GBs to several TBs depending on the size of the dataset.

For training, the following hardware requirements may be needed:

- A powerful GPU with at least 8GB of memory is recommended for training the GAN model efficiently. In general, the more memory the GPU has, the larger the batch size that can be used during training, which can improve the training speed. The experiments are generally carried out on two Intel Xeon machines with GPUs. One of the machines has a Nvidia GeForce GTX 750 Ti and 8 GB RAM while the other has Nvidia GeForce GTX 1080 Ti (12GB), ideally.
- A computer with at least 16GB of RAM or more to handle the memory-intensive training process, which involves loading and processing large batches of data.
- A fast CPU with at least 4 cores or more to efficiently process the model updates and manage the training process.
- Access to cloud computing platforms such as Google Cloud Platform, Google colab or Amazon Web Services can also be beneficial for large-scale training of GAN models. These platforms provide access to powerful GPUs and other resources that can significantly reduce the training time.

The hardware requirements for generating images from text using GAN are similar to the ones for training the model. The generation process requires running the trained model on new text inputs and generating corresponding images. This process can also be computationally expensive, especially for high-resolution images.

The hardware requirements for generating images from text are as follows:

1. CPU: A multi-core CPU with a clock speed of at least 2 GHz is recommended.
2. GPU: A high-end GPU with at least 8 GB of VRAM is recommended for faster computation during training and generation. Nvidia GeForce RTX or Titan series GPUs are commonly used.
3. RAM: At least 16 GB of RAM is recommended for preprocessing and training the model.
4. Storage: Sufficient storage space is required for storing the training data, model checkpoints, and generated images.

5. Network: A stable internet connection is required for downloading the required software and libraries and for transferring large files, such as the training dataset.
6. Power supply: A reliable power supply is recommended to avoid data loss or hardware damage during the training and generation process.

It is essential to optimize the hardware configuration and the hyperparameters of the GAN model for efficient training and to minimize the training time.

3.3.2 Software requirements:

Here is a breakdown of each ideal software requirement with context to the text-to-image synthesis:

1. Operating system: The project can be run on various operating systems such as Windows, macOS, and Linux. However, Linux is preferred for its stability and flexibility.
2. Python Compiler: The project requires Python as the main programming language. It is recommended to use the latest version of Python (currently Python 3.x) as it includes several improvements over the previous versions. Python is very flexible because of the frameworks that make the development of websites and applications quick and easy. There are also many deep learning and data-processing frameworks available in Python, making training of the GAN easier.
3. TensorFlow: TensorFlow is the main deep learning library used for the implementation of GAN. It provides efficient and flexible tools for building and training neural networks. Keras is a high-level Python API for deep learning model development, training, and deployment, and we will be using the Tensorflow back-end for it. Keras having a Python API means it easily interacts with our web-based components as well. We also have more experience with Keras compared to other frameworks.
4. NumPy: NumPy is a fundamental library in Python for scientific computing, providing support for array operations, mathematical functions, and linear algebra. It is extensively used for data processing in GAN for handling and manipulating large arrays of data.
5. Flask is a Python oriented web framework. It simplifies many aspects of web development and its integration with WTForms and Jinja2 allows for simpler creation of HTML via templates.
6. Git: Git is a version control system used for software development. It is used to manage the codebase of the project and facilitate collaboration among team members.

7. CUDA: CUDA is a parallel computing platform and programming model developed by NVIDIA. It is used for GPU acceleration in deep learning applications, including GAN.
8. cuDNN: cuDNN is a GPU-accelerated library of primitives for deep neural networks developed by NVIDIA. It provides support for various operations in deep learning, including convolution and pooling.
9. PyCharm: PyCharm is an integrated development environment (IDE) for Python. It provides a powerful set of tools for code development, debugging, and testing.

These software requirements are crucial for the preprocessing and training of GAN in the text-to-image synthesis project.

4. SYSTEM DESIGN AND ANALYSIS

4.1 Problem Statement AND Conceptual Model

Since computers are deterministic, true randomness cannot be generated. However, it is possible to create algorithms that produce sequences of numbers that exhibit properties similar to those of random sequences. These algorithms can generate a sequence of elements that roughly conforms to a uniform random distribution between 0 and 1 by using a pseudorandom generator. This basic approach can be extended to create more complex random variables. We have utilized this concept to develop an image generation process that produces images that are difficult to distinguish as real or fake. A general semantical data flow diagram is a graphical representation of the flow of data through a system. In the context of the text-to-image synthesis project using GAN, the general semantical data flow diagram would represent the flow of data through the various stages of the process, from input to output.

Here is a high-level overview of the general semantical data flow processes for the project:

1. **Input Data:** The input data for the system consists of textual descriptions of images. These descriptions can be obtained from various sources, such as online databases, user-generated content, or custom datasets.
2. **Text Preprocessing:** The textual data is preprocessed to extract meaningful features such as keywords and sentence structure. This step involves tokenization, part-of-speech tagging, and parsing of the text.
3. **Text Encoder:** The preprocessed text is passed through an encoder network that transforms it into a latent space representation that captures the semantic meaning of the text.
4. **Image Generator:** The latent space representation from the text encoder is then used by the generator network to produce the corresponding image.
5. **Image Discriminator:** The generated image is passed through a discriminator network that assesses its quality and provides feedback to the generator network.

6. Training: The generator and discriminator networks are trained iteratively using adversarial loss functions until the output image quality is satisfactory.
7. Output Image: Once the network is trained, the user can input a textual description, and the system will generate an output image that corresponds to that description.

The data flow diagram for the text-to-image synthesis project using GAN is a high-level representation of the process, and each step may involve multiple sub-steps and more detailed data flow diagrams. Nonetheless, the below diagram provides a basic understanding of how the data flows through the system.

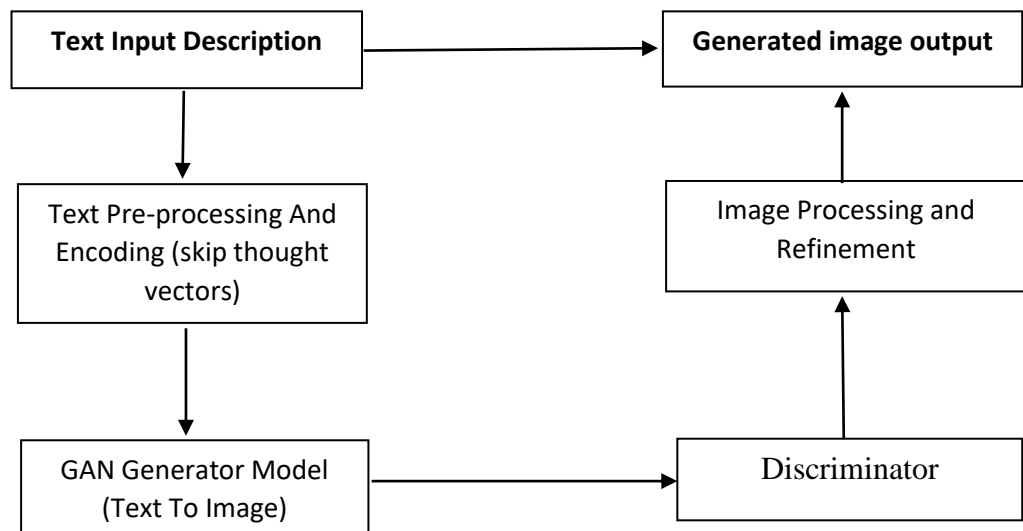


Fig 1: System flowchart of the problem statement

The diagram shows the different components involved in the text-to-image synthesis process. The first component is the text input, which is a textual description of the image that needs to be generated. The text input is then passed through a text preprocessing and encoding module that converts the text into a numerical representation that can be used by the generator model.

The generator model takes the encoded text as input and generates an image output. This output image is then passed through an image postprocessing and refinement module, which refines the generated image to enhance its quality and realism. Finally, the generated image is outputted, and the process is

complete. The model is trained using a discriminator model, which helps the generator model learn to produce more realistic images that closely match the textual description input.

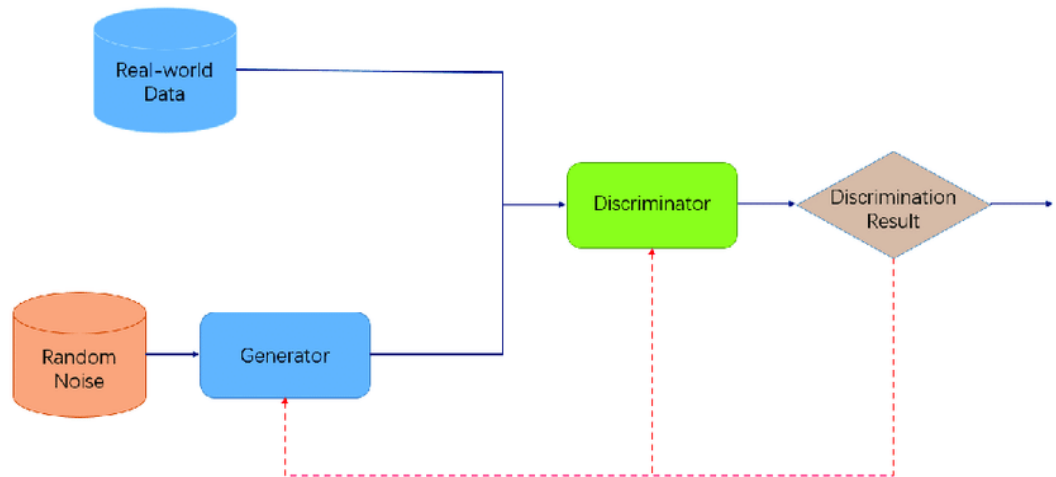


Fig 2: The general semantics flow chart of generative adversarial network (GAN)

4.2 GAN Model OF The Project

We will introduce the necessary symbols before defining them. Considering the dataset $X = \{x(1), \dots, x(m)\}$ composed of m samples, where $x(i)$ is a vector. In the particular case of this report, $x(i)$ is an image encoded in the form of a vector of pixel values. The dataset is, generated by sampling images from an unknown data generating distribution P_r , where r represents the real number. The data generating distribution can be thought of as the hidden distribution of the universe describing a particular phenomenon. A generative model is a model that learns to generate samples from a P_g distribution which predicts

P_r . The P_g distribution is an assumption about the true distribution of the P_r data by maximizing the log-likelihood $E_{X \sim P_r} \log(P_g(x|\theta))$ with respect to the model parameters θ . Intuitively, maximum likelihood learning amounts to placing more probability masses around the regions of X where X has more

examples, and less around the ex-regions with fewer examples. It can be shown that maximizing the log likelihood is equivalent to minimizing the *Kullback-Leibler divergence* $KL(Pr \parallel Pg) = \int Pr \log \frac{Pr}{Pg} dx$ assuming the Pr and density Pg are. A valuable property of this approach is that no knowledge of unknown Pr is required, since the expectation can be approximated with enough samples by the weak law of large numbers. Generative Adversarial Networks (GAN) is another generative model that takes a different approach based on game theory.

GAN comprises of two independent networks. One is called Generator and the other one is called Discriminator. Generator generates synthetic samples given a random noise [sampled from latent space] and the Discriminator is a binary classifier which discriminates between whether the input sample is real [output a scalar value 1] or fake [output scalar value 0].

Fig 3: Samples generated by Generator is termed as fake sample. It takes as input a textual description and generate a rgb image which was described in the textual input. As an example, given - “this flower has a lot of small round pink petals” as input will generate an image of flower having round pink petal as follows:

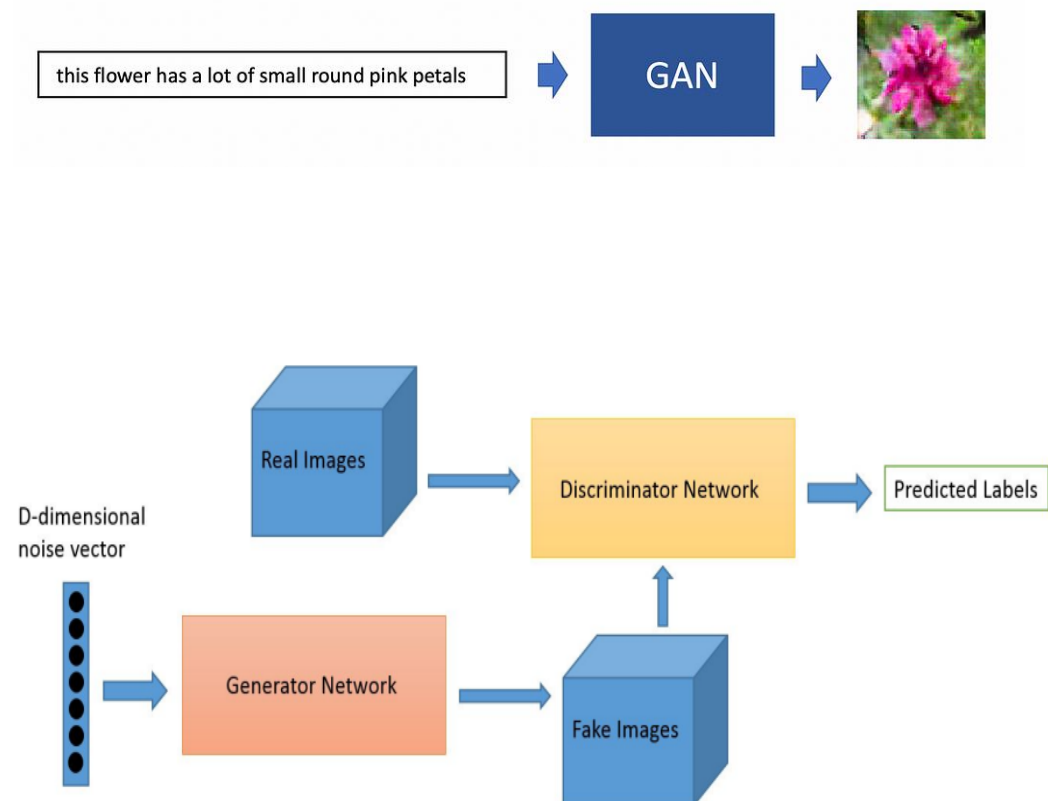


Fig 3: Block diagram of the Generator and Discriminator

4.3 Text Conditioned Auxiliary Classifier Generative Adversarial Network (TAC-GAN)

Our proposed **TAC-GAN model** generates an image of size 128×128 , which conforms to the content of the input text. To train our model, we use the dataset Oxford-102 flowers, where each image has a class tag and at least five text descriptions. For the implementation of TAC-GAN, we use Deep Convolutional Generative Adversarial Networks (DCGAN), the Tensorflow implementation of, where G is modeled as a deconvolutional neural network (fractional striped convolutional), D is modeled as a Convolutional Neural Network (CNN). For our text embedding function ψ , we use Skip-Thought vectors to generate embedding vectors of size N.

In our approach, we introduce a variant of AC-GAN called Text-Conditioned Auxiliary Classifier Generating Adversarial Networks (TAC-GAN) to synthesize images from text. Compared to AC-GAN, we place the images generated by TAC-GAN on text embeddings instead of class labels.

AC-GAN is a variant of the GAN architecture where G packages the generated data onto its class labels and the discriminator performs an ancillary task of classifying synthetic and real data into their respective class labels. In this framework, each generated image is associated with a class label c and a noise vector z , and G uses to generate an image $I_{\text{fake}} = G(c, z)$. The AC-GAN discriminator produces a probability distribution of the sources (false or true), and a probability distribution of the class labels: $DS(I) = P(S | I)$ and $DC(I) = P(C | I)$. The objective function consists of two parts:

- (1) the log-likelihood of the good source LS; and
- (2) the probability log of the correct LC class:

$$LS = E[\log P(S = \text{real} | X_{\text{real}})] + E[\log P(S = \text{false} | X_{\text{false}})]$$

$$LC = E[\log P(C = c | X_{\text{real}})] + E[\log P(C = c | X_{\text{false}})]$$

in During training, D maximizes $LC + LS$, while G minimizes $LC - LS$.

4.4 Modules

4.4.1 Skip through vector(for data preprocessing):

The Skip-thought model is an innovative unsupervised encoder-decoder approach for encoding large bodies of text, which departs from traditional compositional semantics-based techniques but still maintains high quality. During training, the model takes a tuple of sentences (s_{i-1} , s_i , s_{i+1}) as input. The encoder generates a state vector h based on the words in sentence s_i at time t_i . Two decoders are used: one predicts the next word w in sentence s_{i+1} , while the other predicts the previous word w in sentence s_{i-1} based on the current state vector h . The objective function is the sum of the log probabilities of the forward and backward sentences given the encoder representation. The vocabulary used in the RNN-based model (V_{rnn}) is smaller than that used in other representations such as word2vec (V_{w2v}). A conversion function ($f: V_{w2v} \rightarrow V_{rnn}$) can be created using W_v such that v belongs to both V_{rnn} and V_{w2v} . This can be achieved by minimizing the L2 loss to obtain the value of W .

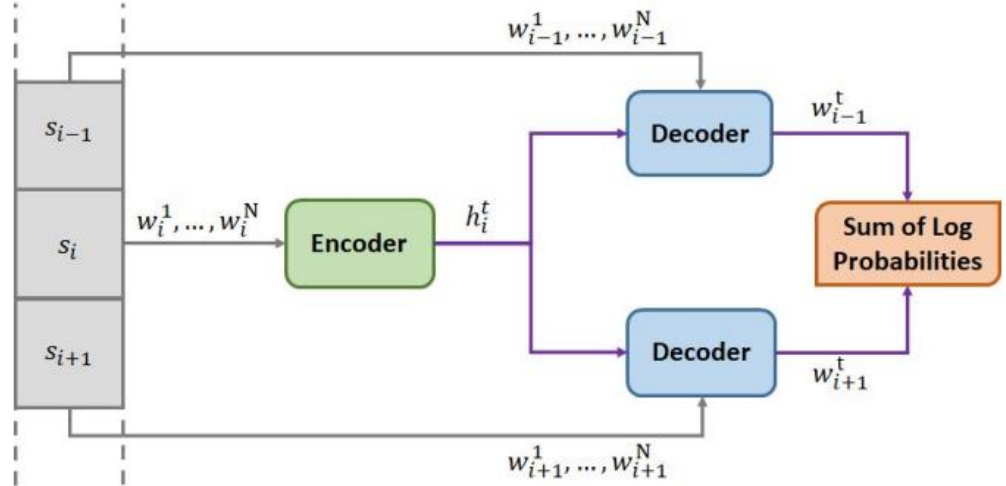


Fig 5: Skip thought vector process diagram

4.4.2 Generative adversarial networks:

Let $X = \{X_i | i = 1, \dots, n\}$ be a dataset, where X_i are the information instances and n is the number of instances within the dataset ($|X| = n$). every records instance is a tuple $X_i = (I_i, T_i, C_i)$, wherein I_i is an picture, $T_i = (t_{1i}, t_{2i}, \dots, t_{ki})$ is a set of k text descriptions of the image, and C_i is the magnificence label to which the photograph corresponds. For education the **TAC-GAN** we randomly choose a text description t_{ji} from T_i and generate a text embedding $\Psi(t_{ji}) \in \mathbb{R}^{N_t}$. We then generate a latent representation $l_{ji} = LG(\Psi(t_{ji}))$ of the textual content embedding, in which LG is a totally linked neural community with N_l neurons. Consequently $l_{ji} \in \mathbb{R}^{N_l}$, in which N_l is a hyperparameter of the model. This representation is then concatenated to a noise vector $z \in [-1, 1]^{N_z}$, creating a vector $z_c \in \mathbb{R}^{N_l+N_z}$. here, N_z is likewise a hyperparameter of the version. z_c is then passed through a completely related layer F_{CG} with $eight * eight * (8 * N_c)$ neurons, where N_c is every other hyperparameter of the model. The output of F_{CG} is finally reshaped right into a convolutional representation \hat{z}_c of form $8 \times eight \times (eight * N_c)$.

4.4.3 GENERATOR:

The Generator network of TAC-GAN is very just like that of the ACGAN. but, instead of feeding the magnificence label to which the synthesized photograph is supposed to pertain, we input the noise vector \hat{z}_c , containing information related to the textual description of the picture. In our version, G is a neural network including a sequence of transposed convolutional layers. It outputs an upscaled image I_f (fake photograph) of shape $128 \times 128 \times 3$.

It generates low-resolution to high decision pics that capture the vast features of the textual content description. The enter to the generator comprises \hat{c} sampled from the latent variable area of the conditioning augmentation and z sampled from an arbitrary distribution $p(z)$. The discriminator does not now recollect whether the input photo is real at this degree; alternatively, it classifies photographs in step with whether or not they match the text description. As a result, there are three input styles as cited above, consisting of real pictures that in shape the input text description, proper pictures that do not match, and generated images. The discriminator's loss characteristic is proven as well as that of the generator.

$$LD_0 = -E(I_0, t) \sim p_{data}[\log D_0(I_0, \phi t)] - E_{z \sim p(z), t \sim p_{data}}[\log(1 - D_0(G_0(z, \hat{c}), \phi t))]$$

$$LD = -\sum_{x \in \chi, z \in \zeta} \log(D(x)) + \log(1 - D(G(z)))$$

$$LG = -\sum_{z \in \zeta} \log(D(G(z)))$$

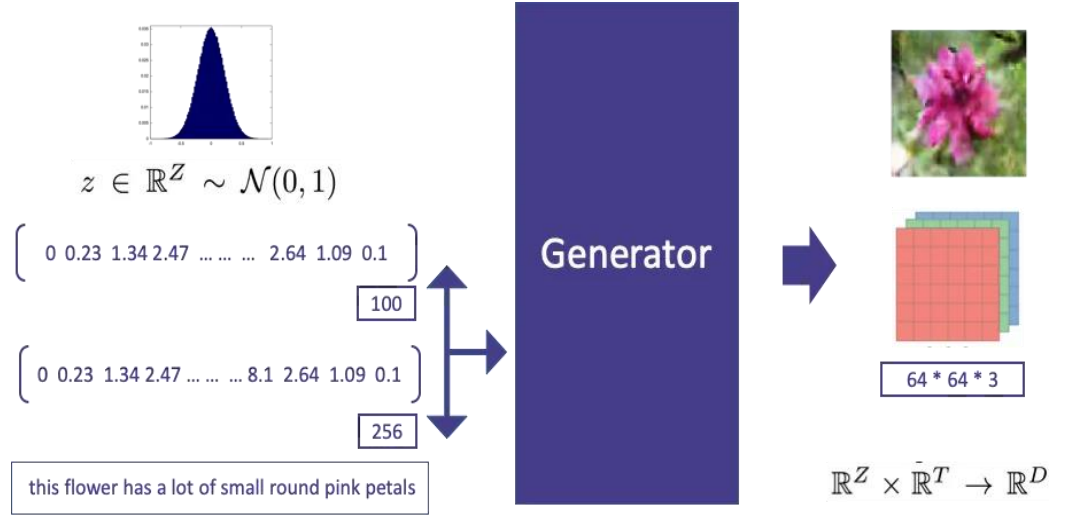


Fig 6: Diagram describing working of the generator

4.4.4 DISCRIMINATOR:

Let I_r denote the actual picture from the dataset corresponding to the text description t_{ji} , and I_w denote every other photo that does not correspond to t_{ji} . Moreover, allow C_r and C_w correspond to the elegance labels of I_r and I_w , respectively. We use I_w to teach the Discriminator to do not forget fake any photo that does not belong to the desired elegance. inside the context of the Discriminator, permit $t_r = t_{ji}$ be the text description of the actual and pretend p_{ix} (notice that the faux picture turned into generated based on the text description of t_r). a new latent representation $l_r = LD(\Psi(t_r))$ for the text embeddings is generated, where LD is any other fully linked neural community with N_l neurons, and for this reason $l_r \in \mathbb{R}^{N_l}$

The Discriminator network is composed by way of a sequence of convolutional layers and receives an image I (any of the photos from A). By passing through the convolutional layers, the image is down sampled into an image MD of length $M \times M \times F$, wherein M and F are hyperparameters of the version. l_r is replicated spatially to form a vector of shape $M \times M \times N_l$, and is concatenated with MD in the F (channels) size, further to what's carried out in . This concatenated vector is then fed to any other convolutional layer with spatial dimension $M \times M$. in the end, fully connected layers $F C_1$ and $F C_2$ are used with 1 and N_c neurons, respectively, along with the sigmoid activation function. $F C_1$ produces a probability distribution DS over the sources (real/fake), while $F C_2$ produces a probability distribution DC over the class labels. Details of the architecture are described in the Figure next. This discriminator design was inspired by the GAN-CLS model proposed by Reed et al. The parameters of LG and LD are trained along with the TAC-GAN.

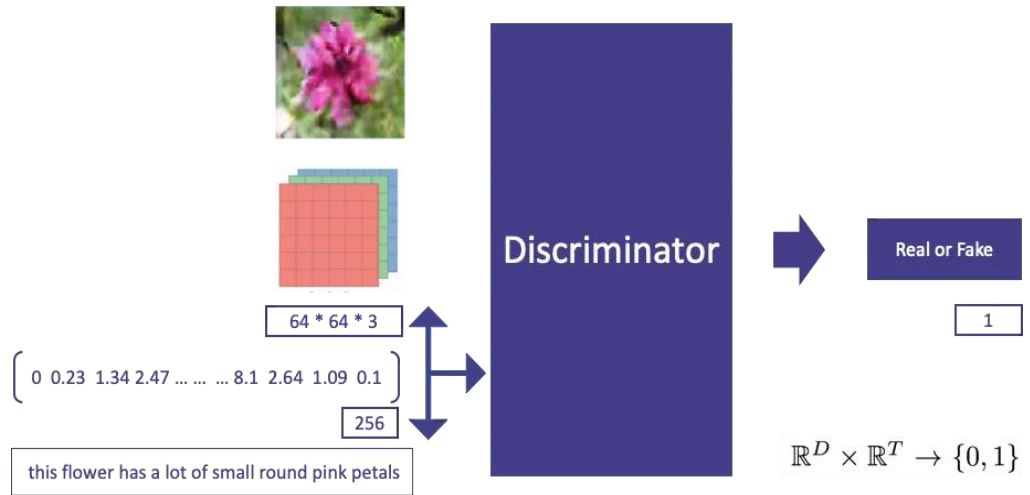


Fig 7: Diagram describing working of the discriminator

4.5 IMPLEMENTATION DETAILS

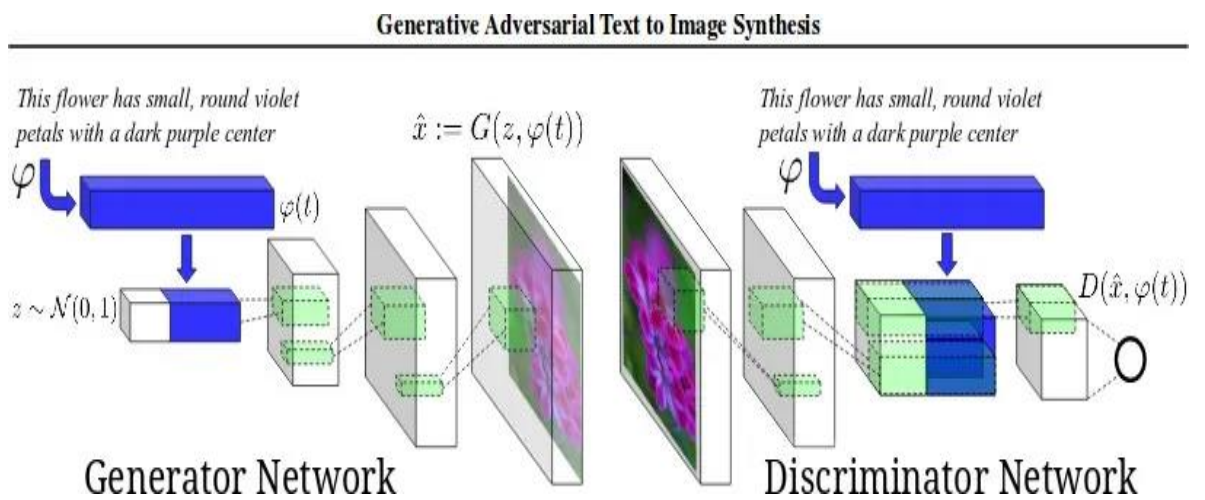


Fig 8: Architecture of GAN

4.5.1 GAN training algorithm:

- 1: Input: minibatch images x , matching text t , mismatching \hat{t} , number of training batch steps S
- 2: for $n = 1$ to S do
- 3: $h \leftarrow \phi(t)$ {Encode matching text description}
- 4: $\hat{h} \leftarrow \phi(\hat{t})$ {Encode mis-matching text description}
- 5: $z \sim N(0, 1)Z$ {Draw sample of random noise}
- 6: $\hat{x} \leftarrow G(z, h)$ {Forward through generator}
- 7: $sr \leftarrow D(x, h)$ {real image, right text}
- 8: $sw \leftarrow D(x, \hat{h})$ {real image, wrong text}
- 9: $sf \leftarrow D(\hat{x}, h)$ {fake image, right text}
- 10: $LD \leftarrow \log(sr) + (\log(1 - sw) + \log(1 - sf))/2$
- 11: $D \leftarrow D - \alpha \partial LD / \partial D$ {Update discriminator}
- 12: $LG \leftarrow \log(sf)$
- 13: $G \leftarrow G - \alpha \partial LG / \partial G$ {Update generator}
- 14: end for

Algorithm summarizes the training procedure. After encoding the textual content, photo and noise (strains 3-five) we generate the faux picture (\hat{x} , line 6). sr indicates the rating of associating a real picture and its corresponding sentence (line 7), sw measures the rating of associating a actual image with an arbitrary sentence (line eight), and sf is the rating of associating a faux picture with its corresponding textual content (line 9). note that we use $\partial LD / \partial D$ to suggest the gradient of D 's objective with appreciate to its parameters, and likewise for G . traces eleven and 13 are intended to signify taking a gradient step to replace community parameters.

4.5.2 IMPLEMENTATION:

Our Generator network is composed by using 3 transposed convolutional layers with 256, 128 and 64 clears out maps, respectively. The output of each layer has a length twice as large as that of the pics fed to them as input. The output of the remaining layer is the produced If used as enter to the Discriminator. D is composed of three convolutional layers with 128, 256, and 384 filter out maps, respectively. The output of the ultimate layer is MD. The kernel length of all the convolutional layers until the era of MD is five \times five. After the concatenation between MD and the spatially replicated lr , the closing

convolutional layer is composed of 512 filter maps of length 1×1 and stride 1. We constantly use identical convolutions, and training is done with assistance of the Adam optimizer.

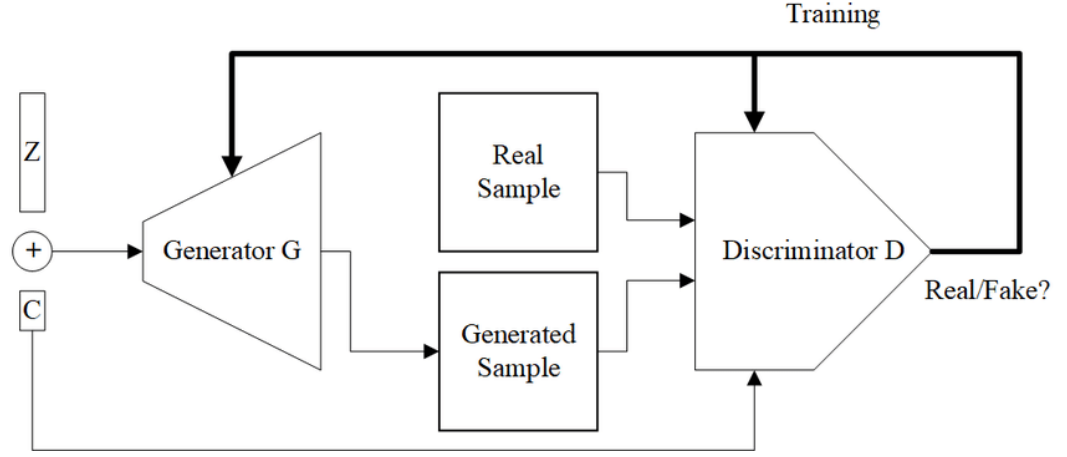


Fig 9: Flow diagram of implementation

4.5.3 TRAINING:

In this phase, we give an explanation for how training is achieved in the TAC-GAN version. We then explain how the loss feature could be effortlessly extended in order that every other doubtlessly useful sort of facts may be leveraged via the model.

Let LDS denote the schooling loss associated with the source of the input (actual, faux or wrong). Then LDS is calculated as a sum of the binary go entropy, denoted with the aid of H below, among the output of the discriminator and the favored cost for every of the images:

$$LDS = H(Ds(Ir, lr), 1) + H(Ds(If, lr), 0) + H(Ds(Iw, lr), 0)$$

Similarly, let LDC denote the training loss related to the class to which the input image is supposed to pertain:

$$LDC = H(Dc(Ir, lr), Cr) + H(Dc(If, lr), Cr) + H(Dc(Iw, lr), Cw)$$

The Discriminator then minimizes the training objective $LDC + LDS$.

For the case of the Generator network, there are no real or wrong photographs to be fed as entered. The training loss, given through $LGC + LGS$, can therefore be described in phrases of most effective the Discriminator's output for the generated picture (LGS), and the anticipated class to which the synthesized photo is anticipated to pertain (LGC):

$$LGS = H(Ds(I_f, I_r), 1)$$

$$LGC = H(Dc(I_f, I_r), C_r)$$

whilst LDS penalizes the move-entropy among $DS(I_f, I_r)$ and zero (making D better at deeming generated pix faux), LGS penalizes the pass-entropy among $DS(I_f, I_r)$ and 1 (approximating the distribution of the generated pix to that of the training statistics).

The training losses for both the G and D is a sum of the losses similar to unique forms of information. In the presence of other types of statistics, you can actually easily expand this sort of loss via including a new thing to the sum. Specially, expect a brand new dataset Y is present, containing, for every real photograph I_r in X , a corresponding vector Q_r containing information, including the presence of certain objects inside the I_r , the region where such info seem, and so on. The Discriminator will be prolonged to output the opportunity distribution $DY(I)$ from any input photo. Let LDY denote the associated Discriminator loss:

$$LDY = H(DY(I_r, I_r), Q_r) + H(DY(I_f, I_r), Q_r) + H(DY(I_w, I_r), Q_w)$$

And LGY denote the related loss for the Generator:

$$LGY = H(DY(I_f, I_r), Q_f)$$

Training could then be achieved with the aid of adding LDY to the Discriminator's loss, and LGY to the Generator's loss. This concept is a generalization of extensions of the GAN framework which have been proposed in previous works.

4.5.4 Loss function:

To train the discriminator network, we use the cross-entropy loss function which should produce a value of 1 when a correct image-sentence pair is given as input, and 0 otherwise. However, this approach may result in the discriminator trying to output 1 even if it actually outputs a value close to 0.9. To address this issue, we apply a technique called "Level-Smoothing" where the modified loss function requires the discriminator to output 0.9 for correct pairs and 0.1 for incorrect pairs. The generator network is trained using binary cross-entropy loss with level-smoothed probabilities, as well as the KL divergence between the conditioning latent variable (predicted by the initial layers of the generator network) and the zero-mean unit variance normal distribution, which is obtained through the reparameterization trick. Additionally, we use a Lagrange multiplier of 2.0 for the KL divergence loss.

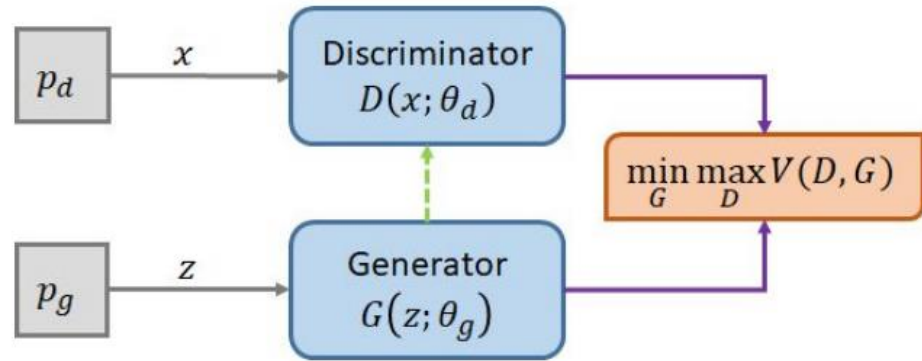


Fig 10: Loss Function

4.5.5 Dataset:

The dataset used for our project is the Oxford-102 flower dataset, which consists of a total of 8,192 images of various flower species. Out of these, 8000 images have been used for training the model, while the remaining 192 images have been used for testing. To improve the accuracy of our model, we have also included 10 captions per image during the training process.

4.6 User Interface

User, task, and environment analysis is a process that involves identifying and analyzing the needs and requirements of the users, tasks they need to perform, and the environment in which they will be working. The goal is to identify the needs and requirements of the users in order to develop a user interface that will meet those needs and provide a positive user experience.

To map the requirements and develop a user interface for text-to-image synthesis using GAN, we need to first identify the target users and their tasks. In this case, the users are those who need to generate realistic images based on text descriptions. The tasks involve inputting text descriptions, selecting parameters for image generation, and reviewing and editing the generated

images. The environment analysis involves identifying the context in which the user interface will be used. This includes the hardware and software platforms that the user interface will be running on, as well as any other environmental factors that may impact the user's experience.

In order to develop the user interface, we will need to consider both external and internal components. The external components include the graphical user interface (GUI) and any other components that the user interacts with directly. The internal components include the algorithms and data structures used to generate the images. Some commonly used data structures in deep learning and image processing projects include arrays, tensors, matrices, and graphs. It is also possible that the project uses custom data structures specific to the implementation of the GAN algorithm or other components of the project.

The architecture of the user interface will involve a front-end and a back-end. The front-end will include the GUI and any other components that the user interacts with directly. The back-end will include the GAN model, which will generate the images based on the text descriptions provided by the user. The user interface will be designed to provide a seamless experience for the user, with intuitive controls and clear feedback on the progress of the image generation process.

We utilized GAN-CLS algorithm to generate images based on the text descriptions using the Oxford-102 Flowers dataset. It provides a diverse range of flower morphology. The user interface takes text input from the user, processes it, and displays the corresponding image that matches the input text description.

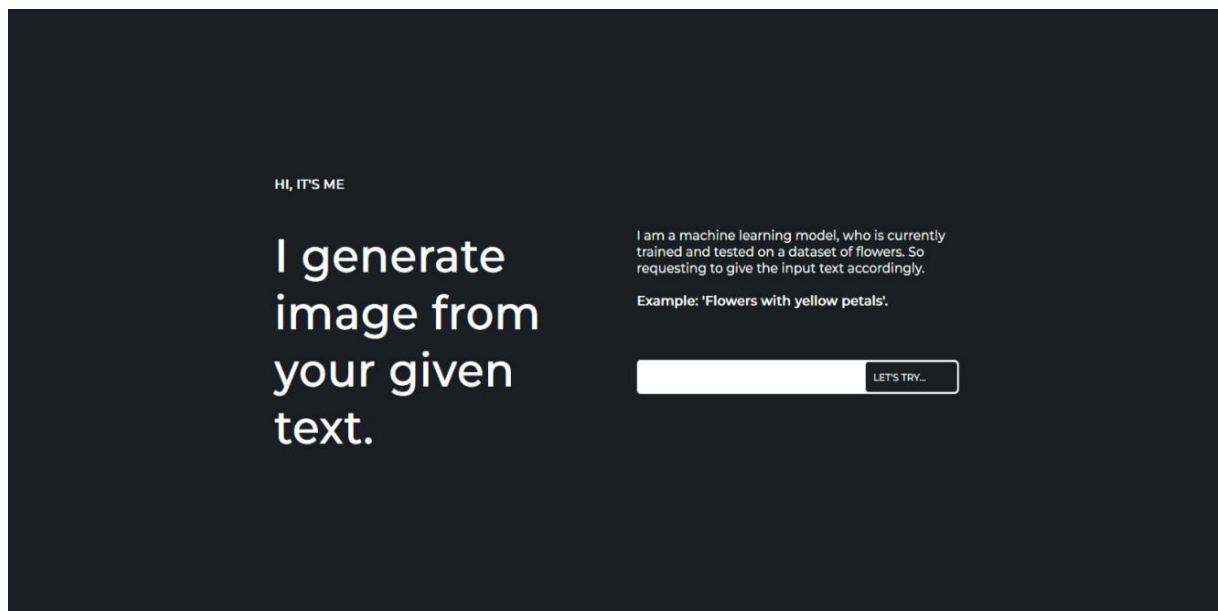
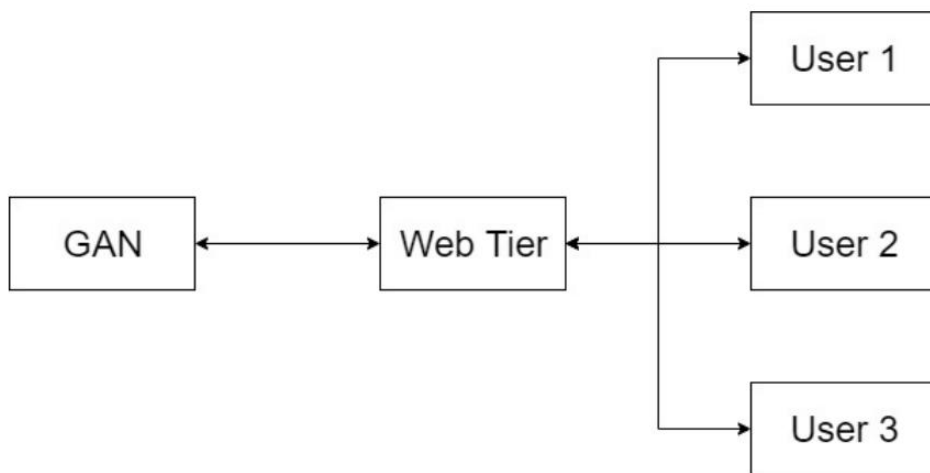


Fig 11: Frontend User Interface

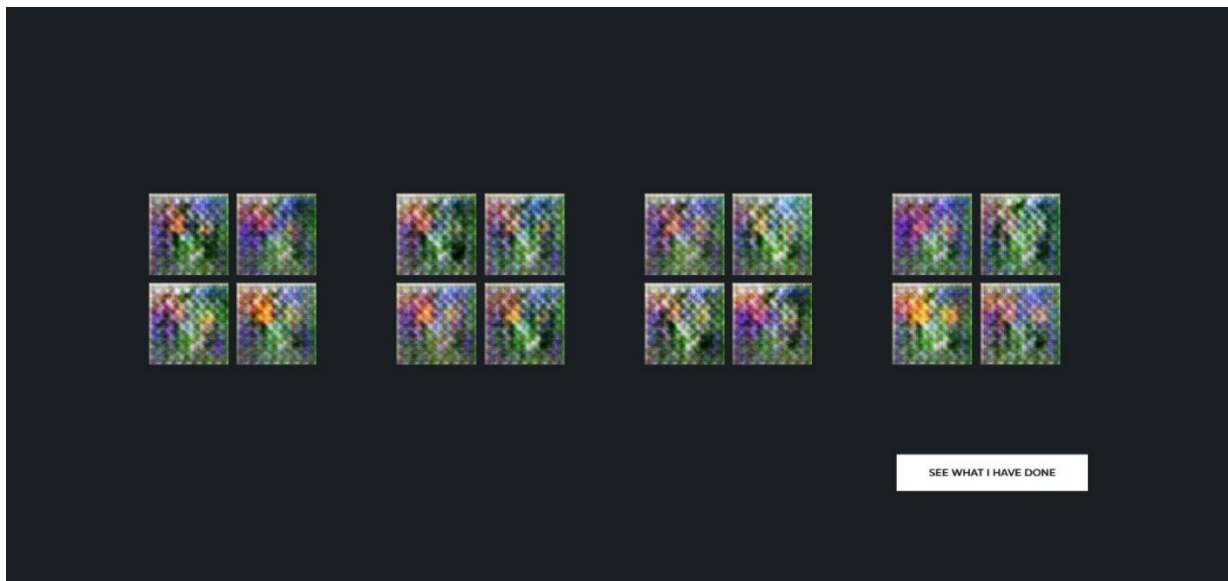


Fig 12: Display screen of the generated images

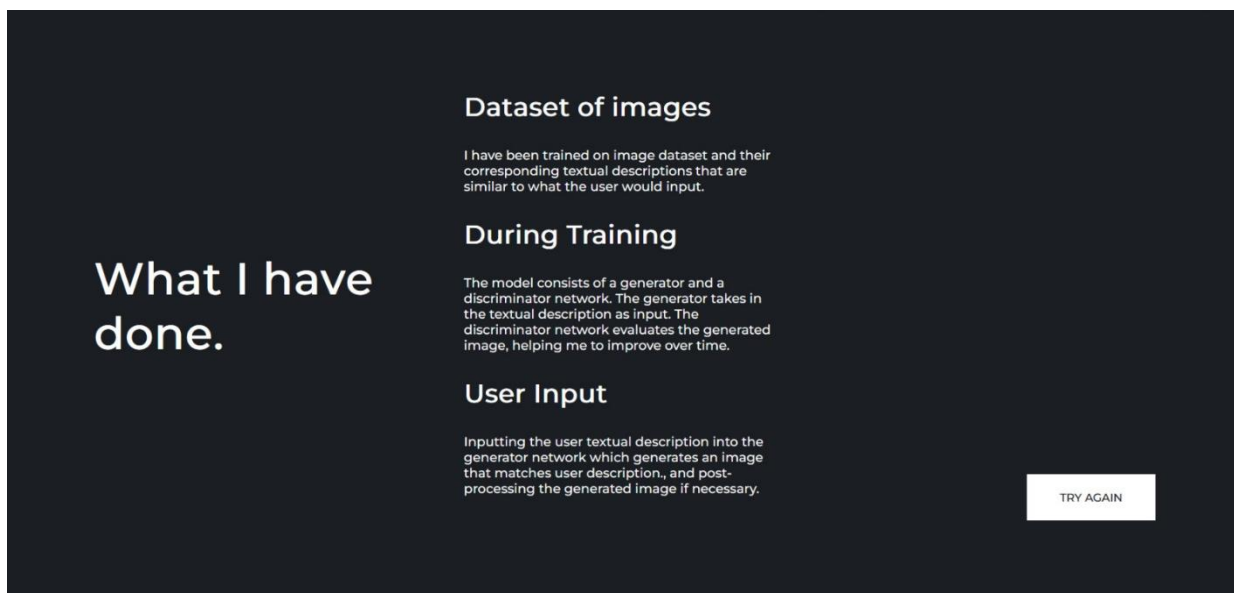


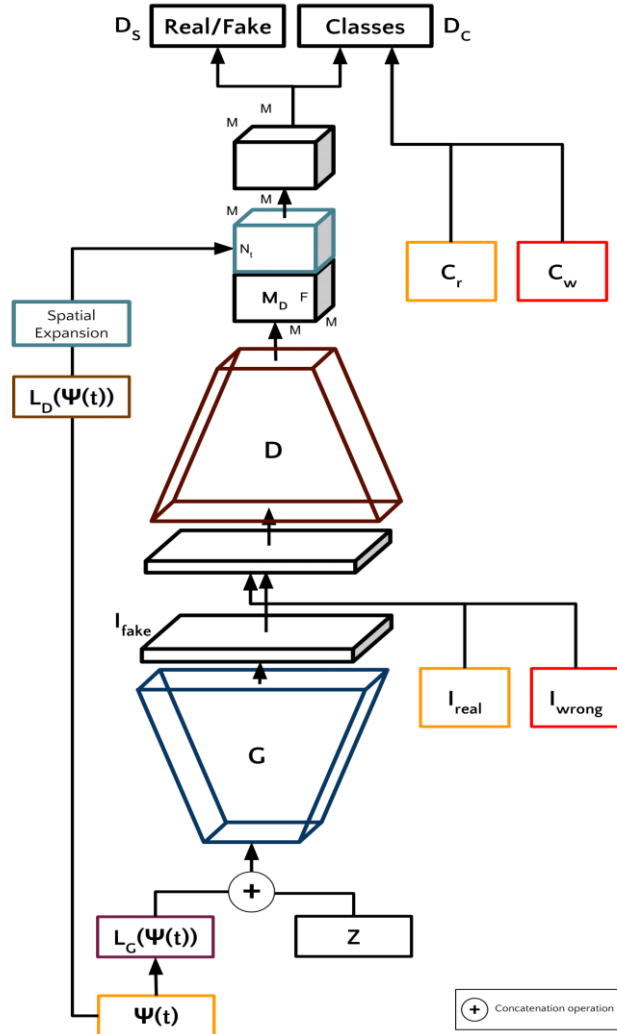
Fig 13: Information display for user

5. IMPLEMENTATION AND TESTING

Text Conditioned Auxiliary Classifier Generative Adversarial Network, is a text to image Generative Adversarial Network (GAN) for synthesizing images from their text descriptions. This GAN builds upon the AC-GAN by conditioning the generated images on a text description instead of on a class label. In the presented GAN model, the input vector of the Generative network is built based on a noise vector and another vector containing an embedded representation of the textual description. While the Discriminator is similar to that of the AC-GAN, it is also augmented to receive the text information as input before performing its classification.

For embedding the textual descriptions of the images into vectors we used skip-thought vectors.

Fig 14: The following is the architecture of our GAN model.



5.1 STEP 1:

The model presented in the paper was trained on the flower's dataset. This To train the TAC-GAN on the flower's dataset, first, download the dataset by doing the following:

1. Download the flower images from here. Extract the 102flowers.tgz file and copy the extracted jpg folder to **Data/datasets/flowers**
2. Download the captions from here. Extract the downloaded file, copy the text_c10 folder and paste it in Data/**datasets/flowers directory**
3. Download the pretrained skip-thought vectors model from here and copy the downloaded files to **Data/skipthoughts**.

5.2 STEP 2:

Data Preprocessing:

Extract the skip-thought features for the captions and prepare the dataset for training by running the following script.

This script will create a set of pickled files in the data directory which will be used during training. The following are the available flags for data preparation:

FLAG	VALUE TYPE	DEFAULT VALUE	DESCRIPTION
data_dir	str	Data	The data directory
dataset	str	flowers	Dataset to use. For Eg., "flowers"

This script appears to be a part of a machine learning project for captioning flower images. The script has several functions for encoding the captions of the images into a vector representation using skip-thought vectors. The vectors are then saved to disk using the pickle module. The script also has some functions for loading the image files and their captions from the disk. The main function of the script is main(), which takes in some command-line arguments and then calls the save_caption_vectors_flowers function to save the vectors to disk. The script then saves the training and validation IDs to disk as well. The script seems to assume that the flower images are organized into folders based on their classes.

```

import os
import argparse
import skipthoughts
import traceback
import pickle
import random

import numpy as np

from os.path import join

def get_one_hot_targets(target_file_path):
    target = []
    one_hot_targets = []
    n_target = 0
    try :
        with open(target_file_path) as f :
            target = f.readlines()
            target = [t.strip("\n") for t in target]
            n_target = len(target)
    except IOError :
        print('Could not load the labels.txt file in the dataset. A '
              'dataset folder is expected in the "data/datasets" '
              'directory with the name that has been passed as an '
              'argument to this method. This directory should contain a '
              'file called labels.txt which contains a list of labels and '
              'corresponding folders for the labels with the same name as '
              'the labels.')
        traceback.print_stack()

    lbl_idxs = np.arange(n_target)
    one_hot_targets = np.zeros((n_target, n_target))
    one_hot_targets[np.arange(n_target), lbl_idxs] = 1

    return target, one_hot_targets, n_target

def one_hot_encode_str_lbl(lbl, target, one_hot_targets):
    """
    Encodes a string label into one-hot encoding
    Example:
        input: "window"
        output: [0 0 0 0 0 0 1 0 0 0 0]
    the length would depend on the number of classes in the dataset. The
    above is just a random example.
    :param lbl: The string label
    :return: one-hot encoding
    """

```

```

        idx = target.index(lbl)
        return one_hot_targets[idx]

def save_caption_vectors_flowers(data_dir, dt_range=(1, 103)) :
    import time

    img_dir = join(data_dir, 'flowers/jpg')
    all_caps_dir = join(data_dir, 'flowers/all_captions.txt')
    target_file_path = os.path.join(data_dir, "flowers/allclasses.txt")
    caption_dir = join(data_dir, 'flowers/text_c10')
    image_files = [f for f in os.listdir(img_dir) if 'jpg' in f]
    print(image_files[300 :400])
    image_captions = { }
    image_classes = { }
    class_dirs = []
    class_names = []
    img_ids = []

    target, one_hot_targets, n_target = get_one_hot_targets(target_file_path)

    for i in range(dt_range[0], dt_range[1]) :
        class_dir_name = 'class_%.5d' % (i)
        class_dir = join(caption_dir, class_dir_name)
        class_names.append(class_dir_name)
        class_dirs.append(class_dir)
        onlyimgfiles = [f[0 :11] + ".jpg" for f in os.listdir(class_dir)
                        if 'txt' in f]
        for img_file in onlyimgfiles:
            image_classes[img_file] = None

        for img_file in onlyimgfiles:
            image_captions[img_file] = []

    for class_dir, class_name in zip(class_dirs, class_names) :
        caption_files = [f for f in os.listdir(class_dir) if 'txt' in f]
        for i, cap_file in enumerate(caption_files) :
            if i%50 == 0:
                print(str(i) + ' captions extracted from' + str(class_dir))
            with open(join(class_dir, cap_file)) as f :
                str_captions = f.read()
                captions = str_captions.split('\n')
                img_file = cap_file[0 :11] + ".jpg"

                # 5 captions per image
                image_captions[img_file] += [cap for cap in captions if len(cap) > 0][0
:5]

                image_classes[img_file] = one_hot_encode_str_lbl(class_name,
                                                                target,
                                                                one_hot_targets)

```

```

model = skipthoughts.load_model()
encoded_captions = {}
for i, img in enumerate(image_captions) :
    st = time.time()
    encoded_captions[img] = skipthoughts.encode(model,
image_captions[img])
    if i%20 == 0:
        print(i, len(image_captions), img)
        print("Seconds", time.time() - st)

img_ids = list(image_captions.keys())

random.shuffle(img_ids)
n_train_instances = int(len(img_ids) * 0.9)
tr_image_ids = img_ids[0 :n_train_instances]
val_image_ids = img_ids[n_train_instances : -1]

pickle.dump(image_captions,
            open(os.path.join(data_dir, 'flowers', 'flowers_caps.pkl'), "wb"))

pickle.dump(tr_image_ids,
            open(os.path.join(data_dir, 'flowers', 'train_ids.pkl'), "wb"))
pickle.dump(val_image_ids,
            open(os.path.join(data_dir, 'flowers', 'val_ids.pkl'), "wb"))

ec_pkl_path = (join(data_dir, 'flowers', 'flower_tv.pkl'))
pickle.dump(encoded_captions, open(ec_pkl_path, "wb"))

fc_pkl_path = (join(data_dir, 'flowers', 'flower_tc.pkl'))
pickle.dump(image_classes, open(fc_pkl_path, "wb"))

def main() :
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type = str, default = 'Data',
                        help = 'Data directory')
    parser.add_argument('--dataset', type=str, default='flowers',
                        help='Dataset to use. For Eg., "flowers"')
    args = parser.parse_args()

    dataset_dir = join(args.data_dir, "datasets")
    if args.dataset == 'flowers':
        save_caption_vectors_flowers(dataset_dir)
    else:
        print('Preprocessor for this dataset is not available.')

if __name__ == '__main__' :
    main()

```


Explanation:

The Python code provides a script that generates encoded vectors of image captions using the Skip-Thoughts model. It also creates one-hot encoded class labels for the images and stores them in separate files.

Here's a detailed explanation of each function:

1. **one_hot_encode_str_lbl(lbl, target, one_hot_targets):** This function takes three parameters: `lbl`, which is the class label to be one-hot encoded, `target`, which is a list of all the class labels, and `one_hot_targets`, which is the one-hot encoding of the class labels. The function returns the one-hot encoding of the `lbl` parameter.

The **one_hot_encode_str_lbl** function in the given code is used to one-hot encode a string label based on a list of possible target values.

Here's how the function works:

The function takes three arguments: `lbl`, `target`, and `one_hot_targets`.

`lbl` is the string label that needs to be one-hot encoded.

`target` is the list of possible target values that `lbl` can take.

`one_hot_targets` is a boolean flag that specifies whether the target values should also be one-hot encoded.

Here are the steps that the function follows to one-hot encode `lbl`:

It creates an empty list `one_hot_lbl` to hold the one-hot encoded label.

It then loops through each possible target value in `target` and checks whether `lbl` matches the target value.

If `lbl` matches the current target value, it appends a 1 to `one_hot_lbl`. Otherwise, it appends a 0.

If `one_hot_targets` is `True`, the function also returns a list of one-hot encoded target values based on `target`.

For example, let's say we have the following label and target values:

python

```
lbl = 'dog'
```

```
target = ['cat', 'dog', 'bird']
```

Calling `one_hot_encode_str_lbl(lbl, target, True)` would return the following one-hot encoded label and target values:

python

```
one_hot_lbl = [0, 1, 0]
```

```
one_hot_targets = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Here, `one_hot_lbl` has a 1 in the second position because the label `lbl` matches the second target value `dog`. `one_hot_targets` is a list of one-hot encoded target values based on `target`, where the first target value `cat` has a 1 in the first position, the second target value `dog` has a 1 in the second position, and the third target value `bird` has a 1 in the third position.

2. **`get_one_hot_targets(target_file_path)`**: This function takes a `target_file_path` parameter, which is the path of the file containing the class labels. The function reads the labels from the file, removes the newlines, and creates a one-hot encoding of the labels. The function returns a tuple of three values: a list of the labels, the one-hot encoded targets, and the number of targets.

The **`get_one_hot_targets(target_file_path)`** function reads the target file located at `target_file_path` and generates a dictionary that maps each unique target to a one-hot encoding vector.

Here's a breakdown of how the function works:

It initializes an empty list called `targets` and reads the lines of the target file using `open()` and `readlines()`.

For each line in the target file, it strips any leading/trailing whitespace and appends the target label to the `targets` list.

It then creates a set of unique targets from the `targets` list using the `set()` function. It creates an empty dictionary called **`one_hot_targets`** to store the one-hot encoding vectors.

For each unique target in the set of targets, it creates a one-hot encoding vector using the **`one_hot_encode_str_lbl()`** function and stores it in the `one_hot_targets` dictionary with the target label as the key.

Finally, the **`one_hot_targets`** dictionary is returned.

Overall, this function generates a dictionary of one-hot encoding vectors that can be used to convert target labels to numerical representations during the training of the machine learning model.

3. **`save_caption_vectors_flowers(data_dir, dt_range,)`**: This is the main function of the script. It takes two parameters: **`data_dir`**, which is the directory where the image files and caption files are located, and **`dt_range`**, which is a tuple of two integers representing the range of directories to be processed. The function reads the image files and captions, creates one-hot encoded class labels for the images, and generates encoded vectors of the captions using the Skip-Thoughts model. The function then shuffles the image IDs, splits them into

training and validation sets, and stores the encoded vectors, class labels, and image IDs in separate pickle files.

The function **save_caption_vectors_flowers(data_dir, dt_range(,))** is used to save the caption vectors for the image captioning task using the flower dataset.

The function takes two arguments:

data_dir: A string representing the directory path where the dataset is stored.

dt_range: A tuple of two integers representing the range of directories containing images to be used for caption vector generation.

The function first loads the pre-trained ResNet-152 model and removes the last fully connected layer from the model. It then loops through the specified range of directories containing images, and for each image in each directory, it generates a feature vector by passing the image through the ResNet model. The feature vector is then passed through a linear layer to generate a caption vector. The caption vector is then saved to a file along with the corresponding image name.

The function saves the generated caption vectors and the corresponding image names as numpy arrays in two separate files in the same directory where the images are located. These saved caption vectors and image names can be used later for training an image captioning model.

5.3 STEP 3:

Describing the GAN Model:

```
class GAN :  
    """  
        OPTIONS  
        z_dim : Noise dimension 100  
        t_dim : Text feature dimension 256  
        image_size : Image Dimension 64  
        gf_dim : Number of conv in the first layer generator 64  
        df_dim : Number of conv in the first layer discriminator 64  
        gfc_dim : Dimension of gen untis for for fully connected layer 1024  
        caption_vector_length : Caption Vector Length 2400  
        batch_size : Batch Size 64  
    """  
  
    def __init__(self, options) :  
        self.options = options
```

```

def build_model(self) :

    print('Initializing placeholder')
    img_size = self.options['image_size']
    t_real_image = tf.placeholder('float32',
[ self.options['batch_size'],
                                img_size, img_size, 3],
                                name = 'real_image')
    t_wrong_image = tf.placeholder('float32',
[ self.options['batch_size'],
                                img_size, img_size, 3],
                                name = 'wrong_image')

    t_real_caption = tf.placeholder('float32',
[ self.options['batch_size'],
self.options['caption_vector_length']],
                                name='real_captions')

    t_z = tf.placeholder('float32', [self.options['batch_size'],
self.options['z_dim']], name='input_noise')

    t_real_classes = tf.placeholder('float32',
[ self.options['batch_size'],
self.options['n_classes']],
                                name='real_classes')

    t_wrong_classes = tf.placeholder('float32',
[ self.options['batch_size'],
self.options['n_classes']],
                                name='wrong_classes')

    t_training = tf.placeholder(tf.bool, name='training')

    print('Building the Generator')
    fake_image = self.generator(t_z, t_real_caption,

    t_training)

    print('Building the Discriminator')
    disc_real_image, disc_real_image_logits, disc_real_image_aux,
\
    disc_real_image_aux_logits = self.discriminator(
    t_real_image, t_real_caption,
self.options['n_classes'],
    t_training)

```

```

        disc_wrong_image, disc_wrong_image_logits,
disc_wrong_image_aux, \
        disc_wrong_image_aux_logits = self.discriminator(
            t_wrong_image, t_real_caption,
self.options['n_classes'],
            t_training, reuse = True)

        disc_fake_image, disc_fake_image_logits,
disc_fake_image_aux, \
        disc_fake_image_aux_logits = self.discriminator(
            fake_image, t_real_caption,
self.options['n_classes'],
            t_training, reuse = True)

        d_right_predictions = tf.equal(tf.argmax(disc_real_image_aux,
1),
            tf.argmax(t_real_classes, 1))
        d_right_accuracy = tf.reduce_mean(tf.cast(d_right_predictions,
            tf.float32))

        d_wrong_predictions =
tf.equal(tf.argmax(disc_wrong_image_aux, 1),
            tf.argmax(t_wrong_classes, 1))
        d_wrong_accuracy =
tf.reduce_mean(tf.cast(d_wrong_predictions,
            tf.float32))

        d_fake_predictions =
tf.equal(tf.argmax(disc_fake_image_aux_logits, 1),
            tf.argmax(t_real_classes, 1))
        d_fake_accuracy = tf.reduce_mean(tf.cast(d_fake_predictions,
            tf.float32))

        tf.get_variable_scope().reuse = False

        print('Building the Loss Function')
        g_loss_1 =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_fake_image_logits,
labels=tf.ones_like(disc_fake_image)))

        g_loss_2 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_fake_image_aux_logits,
labels=t_real_classes))

        d_loss1 = tf.reduce_mean(

```

```

        tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_real_image_logits,
                                labels=tf.ones_like(disc_real_image)))
        d_loss1_1 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_real_image_aux_logits,
                                labels=t_real_classes))
        d_loss2 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_wrong_image_logits,
                                labels=tf.zeros_like(disc_wrong_image)))
        d_loss2_1 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_wrong_image_aux_logits,
                                labels=t_wrong_classes))
        d_loss3 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
logits=disc_fake_image_logits,
                                labels=tf.zeros_like(disc_fake_image)))

        d_loss = d_loss1 + d_loss1_1 + d_loss2 + d_loss2_1 + d_loss3
+ g_loss_2

        g_loss = g_loss_1 + g_loss_2

        t_vars = tf.trainable_variables()
        print('List of all variables')
        for v in t_vars:
            print(v.name)
            print(v)
            self.add_histogram_summary(v.name, v)

        self.add_tb_scalar_summaries(d_loss, g_loss, d_loss1, d_loss2,
d_loss3,
        d_loss1_1, d_loss2_1, g_loss_1, g_loss_2, d_right_accuracy,
        d_wrong_accuracy, d_fake_accuracy)

        self.add_image_summary('Generated Images', fake_image,
                                self.options['batch_size'])

        d_vars = [var for var in t_vars if 'd_' in var.name]
        g_vars = [var for var in t_vars if 'g_' in var.name]

        input_tensors = {

```

```

        't_real_image': t_real_image,
        't_wrong_image': t_wrong_image,
        't_real_caption': t_real_caption,
        't_z': t_z,
        't_real_classes': t_real_classes,
        't_wrong_classes': t_wrong_classes,
        't_training': t_training,
    }

    variables = {
        'd_vars': d_vars,
        'g_vars': g_vars
    }

    loss = {
        'g_loss': g_loss,
        'd_loss': d_loss
    }

    outputs = {
        'generator': fake_image
    }

    checks = {
        'd_loss1': d_loss1,
        'd_loss2': d_loss2,
        'd_loss3': d_loss3,
        'g_loss_1': g_loss_1,
        'g_loss_2': g_loss_2,
        'd_loss1_1': d_loss1_1,
        'd_loss2_1': d_loss2_1,
        'disc_real_image_logits': disc_real_image_logits,
        'disc_wrong_image_logits': disc_wrong_image,
        'disc_fake_image_logits': disc_fake_image_logits
    }

    return input_tensors, variables, loss, outputs, checks

def add_tb_scalar_summaries(self, d_loss, g_loss, d_loss1, d_loss2,
                             d_loss3, d_loss1_1, d_loss2_1, g_loss_1,
                             g_loss_2, d_right_accuracy,
                             d_wrong_accuracy, d_fake_accuracy):

    self.add_scalar_summary("D_Loss", d_loss)
    self.add_scalar_summary("G_Loss", g_loss)
    self.add_scalar_summary("D loss-1 [Real/Fake loss for real
images]",
                             d_loss1)

```

```

images]",
        self.add_scalar_summary("D loss-2 [Real/Fake loss for wrong
                                d_loss2)
images]",
        self.add_scalar_summary("D loss-3 [Real/Fake loss for fake
                                d_loss3)
        self.add_scalar_summary(
            "D loss-4 [Aux Classifier loss for real images]",
d_loss1_1)
        self.add_scalar_summary(
            "D loss-5 [Aux Classifier loss for wrong images]",
d_loss2_1)
        self.add_scalar_summary("G loss-1 [Real/Fake loss for fake
images]",
                                g_loss_1)
        self.add_scalar_summary(
            "G loss-2 [Aux Classifier loss for fake images]",
g_loss_2)
        self.add_scalar_summary("Discriminator Real Image
Accuracy",
                                d_right_accuracy)
        self.add_scalar_summary("Discriminator Wrong Image
Accuracy",
                                d_wrong_accuracy)
        self.add_scalar_summary("Discriminator Fake Image
Accuracy",
                                d_fake_accuracy)

    def add_scalar_summary(self, name, var):
        with tf.name_scope('summaries'):
            tf.summary.scalar(name, var)

    def add_histogram_summary(self, name, var):
        with tf.name_scope('summaries'):
            tf.summary.histogram(name, var)

    def add_image_summary(self, name, var, max_outputs=1):
        with tf.name_scope('summaries'):
            tf.summary.image(name, var,
max_outputs=max_outputs)

    # GENERATOR IMPLEMENTATION based on :
    # https://github.com/carpedm20/DCGAN-
tensorflow/blob/master/model.py
    def generator(self, t_z, t_text_embedding, t_training):

        s = self.options['image_size']
        s2, s4, s8, s16 = int(s / 2), int(s / 4), int(s / 8), int(s / 16)

        reduced_text_embedding = ops.lrelu(

```



```

ops.linear(t_text_embedding, self.options['t_dim'],
'g_embedding'))
z_concat = tf.concat([t_z, reduced_text_embedding], -1)
z_ = ops.linear(z_concat, self.options['gf_dim'] * 8 * s16 * s16,
'g_h0_lin')
h0 = tf.reshape(z_, [-1, s16, s16, self.options['gf_dim'] * 8])
h0 = tf.nn.relu(slim.batch_norm(h0, is_training = t_training,
scope="g_bn0"))

h1 = ops.deconv2d(h0, [self.options['batch_size'], s8, s8,
self.options['gf_dim'] * 4], name = 'g_h1')
h1 = tf.nn.relu(slim.batch_norm(h1, is_training = t_training,
scope="g_bn1"))

h2 = ops.deconv2d(h1, [self.options['batch_size'], s4, s4,
self.options['gf_dim'] * 2], name = 'g_h2')
h2 = tf.nn.relu(slim.batch_norm(h2, is_training = t_training,
scope="g_bn2"))

h3 = ops.deconv2d(h2, [self.options['batch_size'], s2, s2,
self.options['gf_dim'] * 1], name = 'g_h3')
h3 = tf.nn.relu(slim.batch_norm(h3, is_training = t_training,
scope="g_bn3"))

h4 = ops.deconv2d(h3, [self.options['batch_size'], s, s, 3],
name = 'g_h4')
return (tf.tanh(h4) / 2. + 0.5)

```

```

# DISCRIMINATOR IMPLEMENTATION based on :
# https://github.com/carpedm20/DCGAN-
tensorflow/blob/master/model.py
def discriminator(self, image, t_text_embedding, n_classes, t_training,
reuse = False) :
if reuse :
tf.get_variable_scope().reuse_variables()

h0 = ops.lrelu(
ops.conv2d(image, self.options['df_dim'], name =
'd_h0_conv')) # 64

h1 = ops.lrelu(slim.batch_norm(ops.conv2d(h0,
self.options['df_dim'] * 2,
name = 'd_h1_conv'),
reuse=reuse,
is_training = t_training,
scope = 'd_bn1')) # 32

h2 = ops.lrelu(slim.batch_norm(ops.conv2d(h1,
self.options['df_dim'] * 4,

```

```

        name = 'd_h2_conv'),
        reuse=reuse,
        is_training = t_training,
        scope = 'd_bn2')) # 16
h3 = ops.lrelu(slim.batch_norm(ops.conv2d(h2,
        self.options['df_dim'] * 8,
        name = 'd_h3_conv'),
        reuse=reuse,
        is_training = t_training,
        scope = 'd_bn3')) # 8
h3_shape = h3.get_shape().as_list()
# ADD TEXT EMBEDDING TO THE NETWORK
reduced_text_embeddings =
ops.lrelu(ops.linear(t_text_embedding,
        self.options['t_dim'],
        'd_embedding'))
reduced_text_embeddings =
tf.expand_dims(reduced_text_embeddings, 1)
reduced_text_embeddings =
tf.expand_dims(reduced_text_embeddings, 2)
tiled_embeddings = tf.tile(reduced_text_embeddings,
        [1, h3_shape[1], h3_shape[1], 1],
        name = 'tiled_embeddings')

h3_concat = tf.concat([h3, tiled_embeddings], 3, name =
'h3_concat')
h3_new = ops.lrelu(slim.batch_norm(ops.conv2d(h3_concat,
        self.options['df_dim'] * 8,
        1, 1, 1, 1,
        name = 'd_h3_conv_new'),
        reuse=reuse,
        is_training = t_training,
        scope = 'd_bn4')) # 4

h3_flat = tf.reshape(h3_new, [self.options['batch_size'], -1])

h4 = ops.linear(h3_flat, 1, 'd_h4_lin_rw')
h4_aux = ops.linear(h3_flat, n_classes, 'd_h4_lin_ac')

return tf.nn.sigmoid(h4), h4, tf.nn.sigmoid(h4_aux), h4_aux

# This has not been used yet but can be used
def attention(self, decoder_output, seq_outputs, output_size,
time_steps,
        reuse=False) :
    if reuse:
        tf.get_variable_scope().reuse_variables()

```

```

ui = ops.attention(decoder_output, seq_outputs, output_size,
                  time_steps, name = "g_a_attention")

with tf.variable_scope('g_a_attention'):
    ui = tf.transpose(ui, [1, 0, 2])
    ai = tf.nn.softmax(ui, dim=1)
    seq_outputs = tf.transpose(seq_outputs, [1, 0, 2])
    d_dash = tf.reduce_sum(tf.mul(seq_outputs, ai), axis=1)
    return d_dash, ai

```

The code is a TensorFlow implementation of a GAN (Generative Adversarial Network) architecture. The GAN consists of a generator and a discriminator. The generator takes noise and text features as input and generates images, while the discriminator takes an image and text features as input and decides whether the image is real or fake.

The architecture has a number of hyperparameters that can be adjusted by the user, such as the image dimension, the number of convolutional layers in the generator and discriminator, the dimension of the fully connected layer, and the batch size. The loss function includes terms for the generator and discriminator, as well as auxiliary classification losses to classify images based on their class labels. The code is designed to be modular, with separate functions for building the generator and discriminator architectures, which can be easily modified or replaced to experiment with different architectures. The code also includes utility functions for visualizing images and saving summaries to disk.

The two models are trained together in a minimax game, with the generator trying to produce more convincing examples and the discriminator trying to correctly classify them as real or fake.

Let's go through the functions of the GAN model one by one:

1. **__init__(self, options):** This function initializes the GAN model with the given options (hyperparameters).
2. **build_model(self):** This function builds the GAN model by defining the TensorFlow computational graph.
3. **generator(self, input_z, input_txt, training):** This function defines the generator model that takes as input a noise vector `input_z`, a text feature vector `input_txt`, and a boolean flag `training` that specifies whether the model is being trained or tested. The generator model consists of several convolutional layers and transposed convolutional layers that upsample the input noise and text feature vector to generate a new image. The function first passes the `input_z` vector through a fully connected layer

to obtain a tensor of size (batch_size, 64). This tensor is then concatenated with the input_txt vector along the feature dimension, resulting in a tensor of size (batch_size, 74).

The concatenated tensor is then passed through four fully connected layers with ReLU activation, which progressively increase the spatial dimension of the tensor from (batch_size, 74) to (batch_size, 64*64*3). The final output tensor is then reshaped into a 4D tensor of size (batch_size, 64, 64, 3), which corresponds to the generated RGB image.

During training, the generator function is optimized to generate images that can fool the discriminator into classifying them as real. This is achieved by minimizing the binary cross-entropy loss between the discriminator's output on the generated images and a vector of ones, indicating that the generated images should be classified as real.

4. **discriminator(self, input_image, input_txt, n_classes, training, reuse):** This function defines the discriminator model that takes as input an image input_image, a text feature vector input_txt, an integer n_classes that specifies the number of classes in the dataset, a boolean flag training that specifies whether the model is being trained or tested, and a boolean flag reuse that specifies whether to reuse the variables of the discriminator model. The discriminator model consists of several convolutional layers and fully connected layers that down sample the input image and text feature vector to classify it as real or fake.

The input image is first concatenated with a text embedding that has been obtained by passing the input text description through a text encoder. The text embedding is a feature representation of the input text that can be used to condition the discriminator on the text description. The concatenated image-text input is passed through a series of convolutional layers, each followed by a leaky ReLU activation function with a negative slope of 0.2. These convolutional layers learn hierarchical representations of the input image-text pair.

The output of the final convolutional layer is flattened and passed through a dense layer to produce a single scalar output. This output represents the probability score that the input image-text pair is real or fake.

Batch normalization is applied after each convolutional layer to improve training stability.

The reuse argument allows the same set of weights to be reused by the discriminator when it is called multiple times, for example when the discriminator is used to evaluate both real and fake images.

The n_classes argument is used to define the number of classes in a multi-class classification task, if needed.

5. **add_histogram_summary(self, var_name, var):** This function adds a summary to the TensorFlow computational graph that records the histogram of the variable var with name var_name.
6. **add_tb_scalar_summaries(self, d_loss, g_loss, d_loss1, d_loss2, d_loss3, d_loss1_1, d_loss2_1, g_loss_1, g_loss_2, d_right_accuracy, d_wrong_accuracy, d_fake_accuracy):** This function adds several scalar summaries to the TensorFlow computational graph that record the values of various loss functions and accuracy metrics during training.
7. **add_image_summary(self, summary_name, tensor, max_outputs):** This function adds a summary to the TensorFlow computational graph that records max_outputs number of images from the tensor tensor with the name summary_name.

Overall, this GAN model takes in a batch of real images, a batch of wrong images (i.e., images that are not from the real dataset), a batch of text feature vectors, a batch of noise vectors, and a batch of class labels. It then generates a batch of fake images using the generator model and evaluates the real, wrong, and fake images using the discriminator model. The GAN model is trained using the adversarial loss function between the generator and discriminator models, and the model performance is monitored using various loss functions and accuracy metrics.

5.4 STEP 4

TRAINING:

While training, we can monitor samples generated by the model in the Data/training/TAC_GAN/samples directory.

```
# import tensorflow as tf
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

from os.path import join
from Utils import image_processing

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--z_dim', type=int, default=100,
                        help='Noise dimension')

    parser.add_argument('--t_dim', type=int, default=256,
```

```

help='Text feature dimension')

parser.add_argument('--batch_size', type=int, default=64,
                    help='Batch Size')

parser.add_argument('--image_size', type=int, default=128,
                    help='Image Size a, a x a')

parser.add_argument('--gf_dim', type=int, default=64,
                    help='Number of conv in the first
layer gen.')

parser.add_argument('--df_dim', type=int, default=64,
                    help='Number of conv in the first
layer discr.')

parser.add_argument('--caption_vector_length', type=int,
                    default=4800,
                    help='Caption Vector Length')

parser.add_argument('--n_classes', type = int, default = 102,
                    help = 'Number of classes/class labels')

parser.add_argument('--data_dir', type=str, default="Data",
                    help='Data Directory')

parser.add_argument('--learning_rate', type=float, default=0.0002,
                    help='Learning Rate')

parser.add_argument('--beta1', type=float, default=0.5,
                    help='Momentum for Adam
Update')

parser.add_argument('--epochs', type=int, default=200,
                    help='Max number of epochs')

parser.add_argument('--save_every', type=int, default=30,
                    help='Save Model/Samples every
x iterations over '
                    'batches')

parser.add_argument('--resume_model', type=bool, default=False,
                    help='Pre-Trained Model load or
not')

parser.add_argument('--data_set', type=str, default="flowers",
                    help='Dat set: MS-COCO,
flowers')

parser.add_argument('--model_name', type=str, default="TAC_GAN",

```

```

help='model_1 or model_2')

parser.add_argument('--train', type = bool, default = True,
                    help = 'True while training and otherwise')

args = parser.parse_args()

model_dir, model_chkpnts_dir, model_samples_dir,
model_val_samples_dir,\
                                model_summaries_dir =
initialize_directories(args)

datasets_root_dir = join(args.data_dir, 'datasets')
loaded_data = load_training_data(datasets_root_dir, args.data_set,
                                args.caption_vector_length,
                                args.n_classes)

model_options = {
    'z_dim': args.z_dim,
    't_dim': args.t_dim,
    'batch_size': args.batch_size,
    'image_size': args.image_size,
    'gf_dim': args.gf_dim,
    'df_dim': args.df_dim,
    'caption_vector_length': args.caption_vector_length,
    'n_classes': loaded_data['n_classes']
}

# Initialize and build the GAN model
gan = model.GAN(model_options)
input_tensors, variables, loss, outputs, checks = gan.build_model()

d_optim = tf.train.AdamOptimizer(args.learning_rate,
beta1=args.beta1).minimize(loss['d_loss'],

var_list=variables['d_vars'])
g_optim = tf.train.AdamOptimizer(args.learning_rate,
beta1=args.beta1).minimize(loss['g_loss'],

var_list=variables['g_vars'])

global_step_tensor = tf.Variable(1, trainable=False,
name='global_step')
merged = tf.summary.merge_all()
sess = tf.InteractiveSession()

summary_writer = tf.summary.FileWriter(model_summaries_dir,
sess.graph)

```

```

tf.global_variables_initializer().run()
saver = tf.train.Saver(max_to_keep=10000)

if args.resume_model:
    print("Trying to resume training from a previous checkpoint' +
          str(tf.train.latest_checkpoint(model_chkpnts_dir)))
    if tf.train.latest_checkpoint(model_chkpnts_dir) is not None:
        saver.restore(sess,
tf.train.latest_checkpoint(model_chkpnts_dir))
        print('Successfully loaded model. Resuming training.')
    else:
        print('Could not load checkpoints. Training a new
model')
    global_step = global_step_tensor.eval()
    gs_assign_op = global_step_tensor.assign(global_step)
    for i in range(args.epochs):
        batch_no = 0
        while batch_no * args.batch_size + args.batch_size < \
            loaded_data['data_length']:

            real_images, wrong_images, caption_vectors, z_noise,
image_files, \
            real_classes, wrong_classes, image_caps, image_ids = \
get_training_batch(batch_no, args.batch_size,
                    args.image_size, args.z_dim,
                    'train', datasets_root_dir,
                    args.data_set, loaded_data)

            # DISCR UPDATE
            check_ts = [checks['d_loss1'], checks['d_loss2'],
                        checks['d_loss3'], checks['d_loss1_1'],
                        checks['d_loss2_1']]

            feed = {
                input_tensors['t_real_image'].name :
real_images,
                input_tensors['t_wrong_image'].name :
wrong_images,
                input_tensors['t_real_caption'].name :
caption_vectors,
                input_tensors['t_z'].name : z_noise,
                input_tensors['t_real_classes'].name :
real_classes,
                input_tensors['t_wrong_classes'].name :
wrong_classes,
                input_tensors['t_training'].name : args.train
            }

            _, d_loss, gen, d1, d2, d3, d4, d5= sess.run([d_optim,

```



```

        loss['d_loss'], outputs['generator']] + check_ts,
        feed_dict=feed)

        print("D total loss: {}".format(d_loss))
        print("D loss-1 [Real/Fake loss for real images] : {}".format(d1))
        print("D loss-2 [Real/Fake loss for wrong images]: {}".format(d2))
        print("D loss-3 [Real/Fake loss for fake images]: {}".format(d3))
        print("D loss-4 [Aux Classifier loss for real images]: {}".format(d4))
        print("D loss-5 [Aux Classifier loss for wrong images]: {}".format(d5))

        # GEN UPDATE
        _, g_loss, gen = sess.run([g_optim, loss['g_loss'],
                                   outputs['generator']], feed_dict=feed)

        # GEN UPDATE TWICE
        _, summary, g_loss, gen, g1, g2 = sess.run([g_optim,
        merged,
        loss['g_loss'], outputs['generator'], checks['g_loss_1'],
        checks['g_loss_2']], feed_dict=feed)
        summary_writer.add_summary(summary, global_step)
        print("\n\nLOSSES\nDiscriminator Loss: {} \nGenerator
        Loss: {}
        \n\nBatch Number: {} \nEpoch: {} \nTotal Batches per "
        "epoch: {}".format(d_loss, g_loss, batch_no, i,
        int(len(loader.data_loader['image_list']) / args.batch_size)))
        print("\nG loss-1 [Real/Fake loss for fake images] : {}".format(g1))
        print("\nG loss-2 [Aux Classifier loss for fake images]: {}".format(g2))

        global_step += 1
        sess.run(gs_assign_op)
        batch_no += 1
        if (batch_no % args.save_every) == 0 and batch_no != 0:

            print("Saving Images and the Model\n\n")

            save_for_vis(model_samples_dir, real_images,
            gen, image_files,
            image_caps, image_ids)
            save_path = saver.save(sess,
            join(model_checkpoint_dir,
            "latest_model_{}_temp.ckpt".format(
            args.data_set)))

```

```

                                # Getting a batch for validation
                                val_captions, val_image_files, val_image_caps,
val_image_ids = \
    get_val_caps_batch(args.batch_size, loaded_data,
                        args.data_set, datasets_root_dir)

                                shutil.rmtree(model_val_samples_dir)
                                os.makedirs(model_val_samples_dir)

                                for val_viz_cnt in range(0, 4):
                                    val_z_noise = np.random.uniform(-1, 1,
[args.batch_size,
                                                                    args.z_dim])

                                    val_feed = {

                                        input_tensors['t_real_caption'].name : val_captions,
                                                                    input_tensors['t_z'].name :
val_z_noise,
                                                                    input_tensors['t_training'].name :
True
                                                                    }

                                    val_gen = sess.run([outputs['generator']],
                                                        feed_dict=val_feed)

                                    save_for_viz_val(model_val_samples_dir, val_gen,
                                                                    val_image_files,
val_image_caps,
val_image_ids, args.image_size,
val_viz_cnt)

                                # Save the model after every epoch
                                if i % 1 == 0:
                                    epoch_dir = join(model_chkpnts_dir, str(i))
                                    if not os.path.exists(epoch_dir):
                                        os.makedirs(epoch_dir)

                                    save_path = saver.save(sess,
                                                            join(epoch_dir,
                                                                "model_after_{ }_epoch_{ }.ckpt".
                                                                    format(args.data_set, i)))
                                    val_captions, val_image_files, val_image_caps,
val_image_ids = \
                                        get_val_caps_batch(args.batch_size,
loaded_data,
                                                                    args.data_set, datasets_root_dir)

```

```

shutil.rmtree(model_val_samples_dir)
os.makedirs(model_val_samples_dir)

for val_viz_cnt in range(0, 10):
    val_z_noise = np.random.uniform(-1, 1,
[args.batch_size,
                                args.z_dim])

    val_feed = {
        input_tensors['t_real_caption'].name :
val_captions,
        input_tensors['t_z'].name : val_z_noise,
        input_tensors['t_training'].name : True
    }
    val_gen = sess.run([outputs['generator']],
feed_dict=val_feed)
    save_for_viz_val(model_val_samples_dir,
val_gen,
                    val_image_files, val_image_caps,
                    val_image_ids,
args.image_size,
                    val_viz_cnt)

def load_training_data(data_dir, data_set, caption_vector_length, n_classes) :
    if data_set == 'flowers' :
        flower_str_captions = pickle.load(
            open(join(data_dir, 'flowers', 'flowers_caps.pkl'), "rb"))

        img_classes = pickle.load(
            open(join(data_dir, 'flowers', 'flower_tc.pkl'), "rb"))

        flower_enc_captions = pickle.load(
            open(join(data_dir, 'flowers', 'flower_tv.pkl'), "rb"))
        tr_image_ids = pickle.load(
            open(join(data_dir, 'flowers', 'train_ids.pkl'), "rb"))
        val_image_ids = pickle.load(
            open(join(data_dir, 'flowers', 'val_ids.pkl'), "rb"))

        max_caps_len = caption_vector_length
        tr_n_imgs = len(tr_image_ids)
        val_n_imgs = len(val_image_ids)

    return {
        'image_list' : tr_image_ids,
        'captions' : flower_enc_captions,
        'data_length' : tr_n_imgs,
        'classes' : img_classes,
        'n_classes' : n_classes,
        'max_caps_len' : max_caps_len,
        'val_img_list' : val_image_ids,

```

```

        'val_captions' : flower_enc_captions,
        'val_data_len' : val_n_imgs,
        'str_captions' : flower_str_captions
    }

    else :
        raise Exception('No Dataset Found')

def initialize_directories(args):
    model_dir = join(args.data_dir, 'training', args.model_name)
    if not os.path.exists(model_dir):
        os.makedirs(model_dir)

    model_chkpnts_dir = join(model_dir, 'checkpoints')
    if not os.path.exists(model_chkpnts_dir):
        os.makedirs(model_chkpnts_dir)

    model_summaries_dir = join(model_dir, 'summaries')
    if not os.path.exists(model_summaries_dir):
        os.makedirs(model_summaries_dir)

    model_samples_dir = join(model_dir, 'samples')
    if not os.path.exists(model_samples_dir):
        os.makedirs(model_samples_dir)

    model_val_samples_dir = join(model_dir, 'val_samples')
    if not os.path.exists(model_val_samples_dir):
        os.makedirs(model_val_samples_dir)

    return model_dir, model_chkpnts_dir, model_samples_dir, \
        model_val_samples_dir, model_summaries_dir

def save_for_viz_val(data_dir, generated_images, image_files, image_caps,
                    image_ids, image_size, id):

    generated_images = np.squeeze(np.array(generated_images))
    for i in range(0, generated_images.shape[0]) :
        image_dir = join(data_dir, str(image_ids[i]))
        if not os.path.exists(image_dir):
            os.makedirs(image_dir)

        real_image_path = join(image_dir,
                                '{}.jpg'.format(image_ids[i]))
        if os.path.exists(image_dir):
            real_images_255 =
image_processing.load_image_array(image_files[i],

```

```

        image_size, image_ids[i], mode='val')
        imageio.imwrite(real_image_path, real_images_255)

        caps_dir = join(image_dir, "caps.txt")
        if not os.path.exists(caps_dir):
            with open(caps_dir, "w") as text_file:
                text_file.write(image_caps[i]+"\\n")

        fake_images_255 = generated_images[i]
        imageio.imwrite(join(image_dir,
'fake_image_{ }.jpg'.format(id)),
            fake_images_255)

def save_for_vis(data_dir, real_images, generated_images, image_files,
    image_caps, image_ids) :

    shutil.rmtree(data_dir)
    os.makedirs(data_dir)

    for i in range(0, real_images.shape[0]) :
        real_images_255 = (real_images[i, :, :, :])
        imageio.imwrite(join(data_dir,
            '{}_{}.jpg'.format(i, image_files[i].split('/')[1])),
            real_images_255)

        fake_images_255 = (generated_images[i, :, :, :])
        imageio.imwrite(join(data_dir, 'fake_image_{ }.jpg'.format(
            i)), fake_images_255)

    str_caps = '\\n'.join(image_caps)
    str_image_ids = '\\n'.join([str(image_id) for image_id in image_ids])
    with open(join(data_dir, "caps.txt"), "w") as text_file:
        text_file.write(str_caps)
    with open(join(data_dir, "ids.txt"), "w") as text_file:
        text_file.write(str_image_ids)

def get_val_caps_batch(batch_size, loaded_data, data_set, data_dir):

    if data_set == 'flowers':
        captions = np.zeros((batch_size, loaded_data['max_caps_len']))

        batch_idx = np.random.randint(0, loaded_data['val_data_len'],
            size = batch_size)
        image_ids = np.take(loaded_data['val_img_list'], batch_idx)
        image_files = []
        image_caps = []
        for idx, image_id in enumerate(image_ids) :

```

```

        image_file = join(data_dir,
                           'flowers/jpg/' + image_id)
        random_caption = random.randint(0, 4)
        captions[idx, :] = \

loaded_data['val_captions'][image_id][random_caption][
    0 :loaded_data['max_caps_len']]

        image_caps.append(loaded_data['str_captions']
                           [image_id][random_caption])
        image_files.append(image_file)

    return captions, image_files, image_caps, image_ids
else:
    raise Exception('Dataset not found')

def get_training_batch(batch_no, batch_size, image_size, z_dim, split,
                       data_dir, data_set, loaded_data = None) :
    if data_set == 'flowers':
        real_images = np.zeros((batch_size, image_size, image_size,
3))
        wrong_images = np.zeros((batch_size, image_size, image_size,
3))

        captions = np.zeros((batch_size, loaded_data['max_caps_len']))
        real_classes = np.zeros((batch_size, loaded_data['n_classes']))
        wrong_classes = np.zeros((batch_size,
loaded_data['n_classes']))

        cnt = 0
        image_files = []
        image_caps = []
        image_ids = []
        for i in range(batch_no * batch_size,
                        batch_no * batch_size + batch_size) :
            idx = i % len(loaded_data['image_list'])
            image_file = join(data_dir,
                              'flowers/jpg/' +
loaded_data['image_list'][idx])

            image_ids.append(loaded_data['image_list'][idx])

            image_array =
image_processing.load_image_array_flowers(image_file,
                                           image_size)
            real_images[cnt, :, :, :] = image_array

            # Improve this selection of wrong image
            wrong_image_id = random.randint(0,
                                           len(loaded_data['image_list']) - 1)

```

```

        wrong_image_file = join(data_dir,
                                'flowers/jpg/' +
loaded_data['image_list'][
                                wrong_image_id])

        wrong_image_array =
image_processing.load_image_array_flowers(wrong_image_file,
                                            image_size)
        wrong_images[cnt, :, :, :] = wrong_image_array

        wrong_classes[cnt, :] =
loaded_data['classes'][loaded_data['image_list'][
                                wrong_image_id]][0 :loaded_data['n_classes']]

        random_caption = random.randint(0, 4)
        captions[cnt, :] = \
loaded_data['captions'][loaded_data['image_list'][idx]][
                                random_caption][0 :loaded_data['max_caps_len']]

        real_classes[cnt, :] = \
loaded_data['classes'][loaded_data['image_list'][idx]][
                                0 :loaded_data['n_classes']]
        str_cap =
loaded_data['str_captions'][loaded_data['image_list']
                                [idx]][random_caption]

        image_files.append(image_file)
        image_caps.append(str_cap)
        cnt += 1

        z_noise = np.random.uniform(-1, 1, [batch_size, z_dim])
        return real_images, wrong_images, captions, z_noise,
image_files, \
                                real_classes, wrong_classes, image_caps, image_ids
    else:
        raise Exception('Dataset not found')

if __name__ == '__main__':
    main()

```

Explaining the code for training:

1. **load_training_data(data_dir, data_set, caption_vector_length, n_classes)**: This function loads the training data for a specified dataset.

It takes as input the directory where the data is stored, the name of the dataset, the length of the caption vector, and the number of classes in the dataset. The function checks if the specified dataset is "flowers", and loads the relevant data files (captions, image classes, image IDs, etc.) from the corresponding directories. The function then returns a dictionary containing the loaded data, such as the list of image IDs, encoded captions, image classes, and other metadata about the dataset.

2. **initialize_directories(args)**: This function creates a set of directories that will be used to store various outputs from the model during training. It takes as input the command-line arguments passed to the script, which should include the name of the model being trained and the directory where the training data is stored. The function creates a top-level directory for the model, and subdirectories within that for storing checkpoints, samples, summaries, and validation samples. If any of these directories already exist, they are left unchanged.
3. **save_for_viz_val(data_dir, generated_images, image_files, image_caps, image_ids, image_size, id)**: This function saves the generated images and captions for a set of validation images during training. It takes as input the directory where the generated images and captions should be saved, the generated image data (in the form of a numpy array), the filenames and captions for the corresponding real images, the IDs of the images, and some other metadata about the images. The function saves the generated images as JPEG files, along with the original image filenames and captions, in a subdirectory named after the image ID. It also saves the generated captions to a file named "caps.txt" within the same subdirectory.
4. **save_for_vis(data_dir, real_images, generated_images, image_files, image_caps, image_ids)**: This function saves a set of real and generated images and captions to disk for visualization purposes. It takes as input the directory where the images and captions should be saved, as well as the real and generated image data (again, as numpy arrays), filenames and captions for the corresponding real images, and IDs for the images. The function saves the real and generated images as JPEG files in the specified directory, along with the original image filenames and captions. It also saves the captions to a file named "caps.txt" in the same directory, and the image IDs to a file named "ids.txt".

5. **get_val_caps_batch(batch_size, loaded_data, data_set, data_dir):**
This function retrieves a batch of validation captions for a specified dataset. It takes as input the batch size, the loaded training data for the corresponding dataset, the name of the dataset, and the directory where the data is stored. The function checks if the specified dataset is "flowers", randomly selects a batch of image IDs from the validation set, and loads the corresponding captions from the loaded data. It also loads the filenames and captions for the corresponding real images, and returns these as well. If the specified dataset is not "flowers", the function raises an exception.

5.5 STEP 5

GENERATING IMAGE:

```
import model
import pickle
import scipy.misc
import random
import imageio
import os

import tensorflow.compat.v1 as tf
import numpy as np

from os.path import join

def load_training_data(data_dir, data_set, caption_vector_length, n_classes):
    if data_set == 'flowers':
        flower_str_captions = pickle.load(
            open(join(data_dir, 'flowers', 'flowers_caps.pkl'), "rb"))

        img_classes = pickle.load(
            open(join(data_dir, 'flowers', 'flower_tc.pkl'), "rb"))

        flower_enc_captions = pickle.load(
            open(join(data_dir, 'flowers', 'flower_tv.pkl'), "rb"))
        # h1 = h5py.File(join(data_dir, 'flower_tc.hdf5'))
        tr_image_ids = pickle.load(
            open(join(data_dir, 'flowers', 'train_ids.pkl'), "rb"))
        val_image_ids = pickle.load(
            open(join(data_dir, 'flowers', 'val_ids.pkl'), "rb"))
        # caps_new = pickle.load(
        #     open(join('Data', 'enc_text.pkl'), "rb"))
```

```

        # n_classes = n_classes
        max_caps_len = caption_vector_length

        tr_n_imgs = len(tr_image_ids)
        val_n_imgs = len(val_image_ids)

        return {
            'image_list': tr_image_ids,
            'captions': flower_enc_captions,
            'data_length': tr_n_imgs,
            'classes': img_classes,
            'n_classes': n_classes,
            'max_caps_len': max_caps_len,
            'val_img_list': val_image_ids,
            'val_captions': flower_enc_captions,
            'val_data_len': val_n_imgs,
            'str_captions': flower_str_captions,
            # 'text_caps': caps_new
        }

    else:
        raise Exception("This dataset has not been handled yet. "
                        "Contributions are welcome.")

def save_distributed_image_batch(data_dir, generated_images, sel_i, z_i,
                                batch_size=64):
    generated_images = np.squeeze(generated_images)
    folder_name = str(sel_i)
    image_dir = join("static/generated_images/")
    if not os.path.exists(image_dir):
        os.makedirs(image_dir)
    fake_image_255 = generated_images[batch_size-1]
    imageio.imwrite(join(image_dir, '{}.jpg'.format(z_i)),
                    fake_image_255)

def get_caption_batch(loader, data_dir, dataset='flowers',
                      batch_size=64):

    captions = np.zeros((batch_size, loader['max_caps_len']))
    batch_idx = np.random.randint(0, loader['data_length'],
                                   size=batch_size)
    image_ids = np.take(loader['image_list'], batch_idx)
    image_files = []
    image_caps = []
    image_caps_ids = []
    for idx, image_id in enumerate(image_ids):
        image_file = join(data_dir, dataset, 'jpg' + image_id)
        random_caption = random.randint(0, 4)

```

```

        image_caps_ids.append(random_caption)
        captions[idx, :] = \
            loaded_data['captions'][image_id][random_caption][
                0:loaded_data['max_caps_len']]

        image_caps.append(loaded_data['captions']
                           [image_id][random_caption])
        image_files.append(image_file)

    return captions, image_files, image_caps, image_ids, image_caps_ids

z_dim=100 # Noise dimension
t_dim=256 #Text feature dimension
batch_size=64 #batch_size
image_size=128 #Image size a, a*a
gf_dim=64 #Number of conv in the first layer gen.
df_dim=64 #Number of conv in the first layer discr.
caption_vector_length=4800
n_classes=102 #Number of classes/class labels
data_dir="Data" #data directory
learning_rate=0.0002 #learning rate
beta1=0.5 #Momentum for Adam Update.
images_per_caption=30 #The number of images that you want to generateper
text description
data_set="flowers" #Dat set: MS-COCO, flowers
checkpoints_dir="D:\\Downloads Main\\Text-to-Image-Using-GAN-
master\\Text-to-Image-Using-GAN-master\\Data\\training\\TAC-
GAN\\checkpoints\\95" #directory of the checkpoints

datasets_root_dir = join( data_dir, 'datasets')

loaded_data = load_training_data(datasets_root_dir, data_set,
caption_vector_length,
n_classes)
model_options = {
    'z_dim': z_dim,
    't_dim': t_dim,
    'batch_size': batch_size,
    'image_size': image_size,
    'gf_dim': gf_dim,
    'df_dim': df_dim,
    'caption_vector_length': caption_vector_length,
    'n_classes': loaded_data['n_classes']
}

```

```

gan = model.GAN(model_options)
input_tensors, variables, loss, outputs, checks = gan.build_model()

sess = tf.InteractiveSession()
tf.global_variables_initializer().run()

saver = tf.train.Saver(max_to_keep=10000)
print('Loading from ' + checkpoints_dir)
print('Trying to resume model from ' + str(tf.train.latest_checkpoint(
checkpoints_dir)))
if tf.train.latest_checkpoint( checkpoints_dir) is not None:
    saver.restore(sess, tf.train.latest_checkpoint( checkpoints_dir))
    print('Successfully loaded model from ')
else:
    print('Could not load checkpoints. Please provide a valid path
to'
        ' your checkpoints directory')
    exit()

def generate_images():
    caps_new = pickle.load(
        open(join('Data', 'enc_text.pkl'), "rb"))
    loaded_data["text_caps"] = caps_new
    print('Starting to generate images from text descriptions.')
    for sel_i, text_cap in enumerate(loaded_data['text_caps']['features']):

        print("Text idx: {} \n Raw Text: {} \n".format(sel_i, text_cap))
        captions_1, image_files_1, image_caps_1, image_ids_1, \
        image_caps_ids_1 = get_caption_batch(loaded_data,
datasets_root_dir,
        dataset= data_set, batch_size= batch_size)

        captions_1[ batch_size-1, :] = text_cap

        for z_i in range( images_per_caption):
            z_noise = np.random.uniform(-1, 1, [ batch_size,
z_dim])

            val_feed = {
                input_tensors['t_real_caption'].name:
captions_1,
                input_tensors['t_z'].name: z_noise,
                input_tensors['t_training'].name: True
            }

            val_gen = sess.run(
                [outputs['generator']],
                feed_dict=val_feed)
            dump_dir = os.path.join( data_dir,

```

```

                                'images_generated_from_text')
                                save_distributed_image_batch(dump_dir, val_gen,
sel_i, z_i,
                                batch_size)
                                print('Finished generating images from text description')

```

1. **load_training_data(data_dir, data_set, caption_vector_length, n_classes)**: This function loads the training data for the specified dataset. In this particular code, it loads the "flowers" dataset, which contains images of flowers with associated captions. The function returns a dictionary that contains the image IDs, captions, number of classes, and other relevant data.
2. **save_distributed_image_batch(data_dir, generated_images, sel_i, z_i, batch_size=64)**: This function saves a batch of generated images to disk. It takes as input the directory where the images should be saved, the generated images themselves, the index of the current batch, and the index of the current image within the batch.
3. **get_caption_batch(loaded_data, data_dir, dataset='flowers', batch_size=64)**: This function retrieves a batch of captions for use in training the generator. It randomly selects a batch of image IDs, retrieves the corresponding captions, and returns them as a numpy array.
4. **GAN.build_model()**: This function builds the GAN model. It defines the generator and discriminator networks, loss functions, and optimization algorithms. It returns a tuple containing input placeholders, variables, loss functions, output tensors, and other information related to the model.
5. **sess = tf.InteractiveSession()**: This line creates a new TensorFlow session.
6. **tf.global_variables_initializer().run()**: This line initializes all global variables in the TensorFlow graph.
7. **saver = tf.train.Saver(max_to_keep=10000)**: This line creates a TensorFlow saver object, which is used to save and restore models.

8. **saver.restore(sess, tf.train.latest_checkpoint(checkpoints_dir))**: This line restores the latest saved checkpoint of the model. If no checkpoint exists, it does nothing.
9. **GAN.train(sess, input_tensors, variables, loss, output_tensors, checks, loaded_data, data_dir, save_every_epoch=5, max_epochs=600, pretrained_model=None, logs_dir=None)**: This function trains the GAN model using the specified TensorFlow session, input tensors, variables, loss functions, and other parameters. It takes as input the loaded training data, data directory, and various other parameters related to saving and logging the model during training.

5.6 STEP 6

WEBAPP:

Backend development:

```
from flask import Flask, render_template
import os
import pickle
import argparse
import skipthoughts
import sys
import gen_edited

model = skipthoughts.load_model()

# create flask instance
app = Flask(__name__)

@app.route('/encode/<caption>')
def encode_text(caption):
    encoded_captions = { }
    # file_path = os.path.join(args.caption_file)
    dump_path = os.path.join("D:/Downloads Main/Text-to-Image-Using-
GAN-master/Text-to-Image-Using-GAN-master/Data/", 'enc_text.pkl')
    str_captions = caption
    captions = str_captions.split('\n')
    print(captions)
    encoded_captions['features'] = skipthoughts.encode(model, captions)

    pickle.dump(encoded_captions,
```

```

        open(dump_path, "wb"))
    print('Finished extracting Skip-Thought vectors of the given text '
          'descriptions')
    return 'Finished extracting Skip-Thought vectors of the given text
    descriptions'

# app.add_url_rule("/", "generate", gen_edited.generate_images)

@app.route('/generate')
def process_captions():
    gen_edited.generate_images()
    return render_template('pictures.html')

@app.route('/index')
def display_index():
    return render_template('index.html')

@app.route('/what')
def display_what():
    return render_template('what.html')

app.run('localhost',5000)

```

This code is for a Flask web application that can generate images based on text descriptions using a Generative Adversarial Network (GAN). The application takes in text descriptions from users, encodes them using the Skip-Thoughts model, and then generates images based on the encoded captions using the GAN.

Here is a breakdown of the code:

1. Import necessary packages:

- . Flask is the web framework used to create the application.
- . render_template is used to render HTML templates.
- . os is used to interact with the file system.
- . pickle is used to serialize and deserialize Python objects.
- . skipthoughts is a package that provides a pre-trained model for encoding text using the Skip-Thoughts algorithm.
- . sys is used to interact with the system.
- . gen_edited is a module that provides the function for generating images based on the encoded captions.

2. Create a Flask instance:

```
app = Flask(__name__)
```

This creates a new Flask web application.

3. Define a route to encode text:

```
@app.route('/encode/<caption>')  
def encode_text(caption):
```

```
...
```

This defines a route for the application that takes a text description as input.

The caption parameter is the input text description.

The function `encode_text` is called when a user accesses this route.

The function takes in the caption parameter, encodes the text using the Skip-Thoughts model, and then saves the encoded captions to a file using the pickle package.

4. Define a route to generate images:

```
@app.route('/generate')  
def process_captions():
```

This defines a route for the application that generates images based on the encoded text descriptions.

The `process_captions` function is called when a user accesses this route.

The function generates images based on the encoded captions using the `gen_edited` module and then renders an HTML template that displays the generated images.

5. Define a route to display the index page:

```
@app.route('/index')  
def display_index():
```

```
...
```

This defines a route for the application to display the index page.

The `display_index` function is called when a user accesses this route.

The function renders an HTML template that displays the index page.

6. Define a route to display the "what" page:

```
@app.route('/what')  
def display_what():
```

```
...
```

This defines a route for the application to display the "what" page.

The `display_what` function is called when a user accesses this route.
The function renders an HTML template that displays the "what" page.

7. Start the application:

```
app.run('localhost',5000)
```

This starts the Flask application on the local host and port 5000.

Overall, this code defines a Flask web application that allows users to input text descriptions and generate images based on those descriptions using a GAN. The text descriptions are encoded using the Skip-Thoughts model, and the generated images are displayed to the user using HTML templates.

Frontend development

Index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="/static/css/index.css" />
    <script src="/static/js/index.js"></script>
    <title>Hi</title>
  </head>
  <body>
    <div class="main">
      <div class="main--left">
        <h2 class="main--left-heading">HI, IT'S ME</h2>
        <h1 class="main--left-content">
          I generate image from your given text.
        </h1>
      </div>
      <div class="main--right">
        <p class="main--right-p">
          I am a machine learning model, who is currently trained and tested on
          a dataset of flowers. So requesting to give the input text
          accordingly. <br />
        </p>
      </div>
    </div>
  </body>
</html>
```

```

        <br />
        <span class="main--right-example">Example: 'Flowers with yellow
petals'.</span>
    </p>
    <span class="main--right-button">
        <span class="main--right-main" id="try">LET'S TRY...</span>
        <input class="main--right-text" type="text" name="" id="" />
    </span>
</div>
</div>
</body>
</html>

```

Pictures.html:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="/static/css/index.css" />
    <script src="/static/js/index.js"></script>
    <title>Hi</title>
</head>
<body>
    <div class="main">
        <div class="main--left">
            <h2 class="main--left-heading">HI, IT'S ME</h2>
            <h1 class="main--left-content">
                I generate image from your given text.
            </h1>
        </div>
        <div class="main--right">
            <p class="main--right-p">
                I am a machine learning model, who is currently trained and tested on
                a dataset of flowers. So requesting to give the input text
                accordingly. <br />
            <br />
            <span class="main--right-example">Example: 'Flowers with yellow
petals'.</span>
        </p>
    </div>

```

```

    <span class="main--right-button">
      <span class="main--right-main" id="try">LET'S TRY...</span>
      <input class="main--right-text" type="text" name="" id="" />
    </span>
  </div>
</div>
</body>
</html>

```

What.html:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="/static/css/what.css" />
    <script src="../js/index.js"></script>
    <title>Hi</title>
  </head>
  <body>
    <div class="main">
      <div class="main--left">
        <h1 class="main--left-content">What I have done.</h1>
      </div>
      <div class="main--center">
        <div class="main--section">
          <h1 class="main--section-heading">Dataset of images</h1>
          <p class="main--section-body">
            I have been trained on image dataset and their corresponding textual
            descriptions that are similar to what the user would input.
          </p>
        </div>

        <div class="main--section">
          <h1 class="main--section-heading">During Training</h1>
          <p class="main--section-body">
            The model consists of a generator and a discriminator network. The
            generator takes in the textual description as input. The
            discriminator network evaluates the generated image, helping me to
            improve over time.
          </p>
        </div>
      </div>
    </div>
  </body>
</html>

```

```

    </p>
  </div>
  <div class="main--section">
    <h1 class="main--section-heading">User Input</h1>
    <p class="main--section-body">
      Inputting the user textual description into the generator network
      which generates an image that matches user description., and
      post-processing the generated image if necessary.
    </p>
  </div>
</div>
<div class="main--right">
  <span class="main--right-button">
    <a href=" ../index">TRY AGAIN</a>
  </span>
</div>
</div>
</body>
</html>

```

CSS FILES:

Index.css:

```

@font-face {
  font-family: Montserrat;
  src: url("/static/fonts/Montserrat-Medium.ttf");
}

:root {
  --bg-primary: #1b1f24;
  --primary-color: white;
}
*,
*::before,
*::after {
  padding: 0;
  margin: 0;
}
body {
  display: flex;
  align-items: center;

```

```

justify-content: center;
background-color: var(--bg-primary);
color: var(--primary-color);
height: 100vh;
font-family: Montserrat;
}
.main {
display: flex;
flex-direction: row;
justify-content: baseline;
gap: 60px;
/* padding: 300px; */
}
.main--left {
width: 400px;
}
.main--right {
width: 400px;
}
.main--right-p {
margin-top: 60px;
}
.main--right-main {
content: "LET'S TRY...";
position: absolute;
font-size: 14px;
top: 0px;
left: 0px;
z-index: 1;
display: block;
font-size: 12px;
/* font-weight: bold; */
background-color: var(--primary-color);
cursor: pointer;
color: var(--bg-primary);
padding: 10px;
width: 70%;
max-height: 100%;
border-radius: 5px;
}
.main--right-button {
position: relative;
font-size: 14px;
display: block;

```

```

margin-top: 60px;
background-color: var(--primary-color);
cursor: pointer;
color: var(--bg-primary);
padding: 20px;
width: 70%;
border-radius: 5px;
}

.main--right-main-move {
width: 30%;
left: 90%;
background-color: var(--bg-primary);
color: var(--primary-color);
border: 3px solid var(--primary-color);
transition: all 0.1s linear;
}

.main--left-heading {
font-size: medium;
margin-bottom: 40px;
}

.main--right-text {
position: absolute;
top: 0;
left: 0;
height: 100%;
width: 100%;
z-index: 0;
background-color: var(--primary-color);
cursor: pointer;
color: var(--bg-primary);
width: 70%;
border-radius: 5px;
outline: none;
border: none;
padding-left: 10px;
}

.main--left-content {
font-size: 55px;
}

.main--right-example {
font-weight: bold; }

```

Pictures.css:

```
@font-face {
  font-family: Montserrat;
  src: url("../fonts/Montserrat-Medium.ttf");
}

:root {
  --bg-primary: #1b1f24;
  --primary-color: white;
}
*,
*::before,
*::after {
  padding: 0;
  margin: 0;
}
body {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  background-color: var(--bg-primary);
  color: var(--primary-color);
  height: 100vh;
  gap: 50px;
  font-family: Montserrat;
}
.gallery {
  display: flex;
  flex-direction: row;
  justify-content: baseline;
  gap: 60px;
  /* padding: 300px; */
}
.gallery--section img {
  height: 100px;
  width: auto;
}
.gallery--section {
  display: flex;
  gap: 10px;
  flex-wrap: wrap;
```

```

    width: 250px;
  }
.button {
  position: absolute;
  right: 30px;
  font-size: 12px;
  display: block;
  margin-top: 60px;
  background-color: var(--primary-color);
  cursor: pointer;
  color: var(--bg-primary);
  padding: 15px 40px;
  font-weight: bold;
  width: fit-content;

  /* border-radius: 5px; */
}
a {
  text-decoration: none;
  color: var(--bg-primary);
}

.button--main {
  width: 80%;
  position: relative;
  /* text-align: right; */
}

```

What.css:

```

@font-face {
  font-family: Montserrat;
  src: url("../fonts/Montserrat-Medium.ttf");
}

:root {
  --bg-primary: #1b1f24;
  --primary-color: white;
}
*,
*::before,
*::after {

```



```
padding: 0;
margin: 0;
}
body {
display: flex;
align-items: center;
justify-content: center;
background-color: var(--bg-primary);
color: var(--primary-color);
height: 100vh;
font-family: Montserrat;
}
.main {
display: flex;
flex-direction: row;
justify-content: baseline;
gap: 60px;
/* padding: 300px; */
}
.main--left {
display: flex;
align-items: center;
width: 400px;
}
.main--right {
display: flex;
align-items: end;
justify-content: flex-end;
width: 400px;
}
.main--right-p {
margin-top: 60px;
}
.main--right-main {
content: "LET'S TRY...";
position: absolute;
font-size: 14px;
top: 0px;
left: 0px;
z-index: 1;
display: block;
font-size: 12px;
/* font-weight: bold; */
background-color: var(--primary-color);
```

```

    cursor: pointer;
    color: var(--bg-primary);
    padding: 10px;
    width: 70%;
    max-height: 100%;
    border-radius: 5px;
}
.main--right-button {
    position: relative;
    font-size: 14px;
    display: block;
    margin-top: 60px;
    background-color: var(--primary-color);
    cursor: pointer;
    color: var(--bg-primary);
    padding: 20px;
    width: 30%;
    text-align: center;
    /* border-radius: 5px; */
}

.main--right-main-move {
    width: 30%;
    left: 90%;
    background-color: var(--bg-primary);
    color: var(--primary-color);
    border: 3px solid var(--primary-color);
    transition: all 0.1s linear;
}

.main--left-content {
    font-size: 55px;
}

.main--right-example {
    font-weight: bold;
}

.main--center {
    width: 400px;
}

.main--section {
    margin-bottom: 30px;
}

.main--section-heading {

```

```
margin-bottom: 30px;
}
a {
  text-decoration: none;
  color: var(--bg-primary);
}
```

JavaScript file:

```
window.onload = function () {
  const try_button = document.getElementById("try");
  try_button.addEventListener("click", () => {
    try_button.classList.add("main--right-main-move");
  });
};
```

5.7 TESTING

Evaluating generative models, like GANs, have always been a challenge. While no standard evaluation metric exists, recent works have introduced a lot of new metrics. However, not all of these metrics are useful for evaluating GANs. The Inception Score is a widely used metric for evaluating the quality of GAN generated images. It measures the quality and diversity of the generated images by considering the accuracy of an Inception-v3 model in predicting the class labels of the generated images.

The Inception Score measures the quality of GAN generated images by evaluating how well an Inception-v3 model, which is a deep neural network pre-trained on a large dataset of real-world images, can classify the generated images. The intuition behind this metric is that if the generated images are of high quality and diversity, then a well-trained classifier should be able to classify them with high accuracy.

The calculation of the inception score involves first using the inception v3 model to calculate the conditional probability for each image ($p(y|x)$). The marginal probability is then calculated as the average of the conditional probabilities for the images in the group ($p(y)$). The KL divergence between $p(y)$ and $p(y|x)$ is then calculated and multiplied by $p(y|x)$. Finally, the score is calculated as the exponential of this value.

Here is the algorithm for calculating the Inception Score:

1. Generate a set of N images using the GAN model.
2. Preprocess the images by resizing them to 299×299 and scaling pixel values to the range $[-1, 1]$.
3. Feed the pre-processed images into the Inception-v3 network to obtain the softmax probabilities for each image. The output of the network is a vector of size 1000, where each element represents the probability of the image belonging to a specific class.
4. Calculate the marginal distribution $P(y)$ of the Inception-v3 predictions across all images. This is done by averaging the softmax probabilities across all images.
5. For each generated image, calculate the conditional distribution $P(y|x)$, where x is the generated image and y is the predicted class label. This is also done by feeding the image into the Inception-v3 network and obtaining the softmax probabilities.
6. Calculate the KL divergence between the conditional distribution $P(y|x)$ and the marginal distribution $P(y)$ for each generated image. This measures how different the distribution of predicted class labels is for each generated image compared to the overall distribution.
7. Calculate the Inception Score as the exponential of the average KL divergence across all generated images:

$$\text{Inception Score} = \exp(E[\text{KL}(P(y|x) \parallel P(y))])$$

where $E[\]$ represents the expected value over all generated images.

Note that a higher Inception Score indicates that the generated images are of higher quality and diversity, while a lower Inception Score indicates poor quality or lack of diversity in the generated images.

Model	Inception Score
TAC-GAN	3.45 +- 0.05
StackGan	3.20 +- .01
GAN-INT-CLS	2.66 +- .03

Inception Score of the generated samples on the Oxford-102 dataset.

While the Inception Score is a widely used metric for evaluating the quality of GAN generated images, it has some limitations. For example, it does not take into account the visual quality of the generated images and can be biased towards generating images that are similar to the training data. Therefore, it is often used in combination with other evaluation metrics to provide a more comprehensive evaluation of GAN generated images.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os.path
import sys
import tarfile

import numpy as np
from six.moves import urllib
import tensorflow as tf
import glob
import scipy.misc
import math
import sys

MODEL_DIR = '/tmp/imagenet'
DATA_URL =
'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-
05.tgz'
softmax = None

# Call this function with list of images. Each of elements should be a
# numpy array with values ranging from 0 to 255.
# This function takes two arguments: a list of images and an optional
# argument called "splits" which determines how many times to split the
# predictions when calculating the inception score.
def get_inception_score(images, splits=10):
```

#check that the input images are valid by ensuring that they are stored in a list, that they are numpy arrays with three dimensions (height, width, channels), and that their pixel values fall within a certain range.

```
assert(type(images) == list)
assert(type(images[0]) == np.ndarray)
assert(len(images[0].shape) == 3)
assert(np.max(images[0]) > 10)
assert(np.min(images[0]) >= 0.0)
```

convert each image to a float32 data type and add an extra dimension to it so that it can be fed through the Inceptionv3 model.

```
inps = []
for img in images:
    img = img.astype(np.float32)
    inps.append(np.expand_dims(img, 0))
bs = 100
with tf.Session() as sess:
    preds = []
    n_batches = int(math.ceil(float(len(inps)) / float(bs)))
    for i in range(n_batches):
        sys.stdout.write(".")
        sys.stdout.flush()
        inp = inps[(i * bs):min((i + 1) * bs, len(inps))]
        inp = np.concatenate(inp, 0)
        pred = sess.run(softmax, {'ExpandDims:0': inp})
        preds.append(pred)
    preds = np.concatenate(preds, 0)
```

#calculate the inception score

```
scores = []
for i in range(splits):
    part = preds[(i * preds.shape[0] // splits):((i + 1) * preds.shape[0] // splits),:]
    kl = part * (np.log(part) - np.log(np.expand_dims(np.mean(part, 0), 0)))
    kl = np.mean(np.sum(kl, 1))
    scores.append(np.exp(kl))
return np.mean(scores), np.std(scores)
```

This function is called automatically.

```
def _init_inception():
```

#This function initializes a pre-trained Inceptionv3 model.

```
    global softmax
```

#declares a global variable called “softmax” & check if a directory called “inception” exists in the current working directory. If it doesn’t exist, it creates one.

```
    if not os.path.exists(MODEL_DIR):
        os.makedirs(MODEL_DIR)
    filename = DATA_URL.split('/')[-1]
```

```

filepath = os.path.join(MODEL_DIR, filename)
if not os.path.exists(filepath):
    def _progress(count, block_size, total_size):
        sys.stdout.write("\r>> Downloading %s %.1f%%" % (
            filename, float(count * block_size) / float(total_size) * 100.0))
        sys.stdout.flush()
    filepath, _ = urllib.request.urlretrieve(DATA_URL, filepath, _progress)
    print()
    statinfo = os.stat(filepath)
    print('Succesfully downloaded', filename, statinfo.st_size, 'bytes.')
    tarfile.open(filepath, 'r:gz').extractall(MODEL_DIR)

#extracts the contents of the downloaded tar file into the “inception” directory.
with tf.gfile.FastGFile(os.path.join(
    MODEL_DIR, 'classify_image_graph_def.pb'), 'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    _ = tf.import_graph_def(graph_def, name=")

# Works with an arbitrary minibatch size.
with tf.Session() as sess:
    pool3 = sess.graph.get_tensor_by_name('pool_3:0')
    ops = pool3.graph.get_operations()
    for op_idx, op in enumerate(ops):
        for o in op.outputs:
            shape = o.get_shape()
            shape = [s.value for s in shape]
            new_shape = []
            for j, s in enumerate(shape):
                if s == 1 and j == 0:
                    new_shape.append(None)
                else:
                    new_shape.append(s)
            o._shape = tf.TensorShape(new_shape)
    w = sess.graph.get_operation_by_name("softmax/logits/MatMul").inputs[1]
    logits = tf.matmul(tf.squeeze(pool3), w)
    softmax = tf.nn.softmax(logits)

if __name__ == '__main__':
    if softmax is None:
        _init_inception()

    def get_images(filename):
        return scipy.misc.imread(filename)
    filenames = glob.glob(os.path.join('./data', '*.png'))
    images = [get_images(filename) for filename in filenames]
    print(len(images))
    print(get_inception_score(images))

```

6. RESULTS AND DISCUSSION

This result section provides a comprehensive evaluation of the performance of the model, i.e., the qualitative results. The following are some test results generated for different user input text.

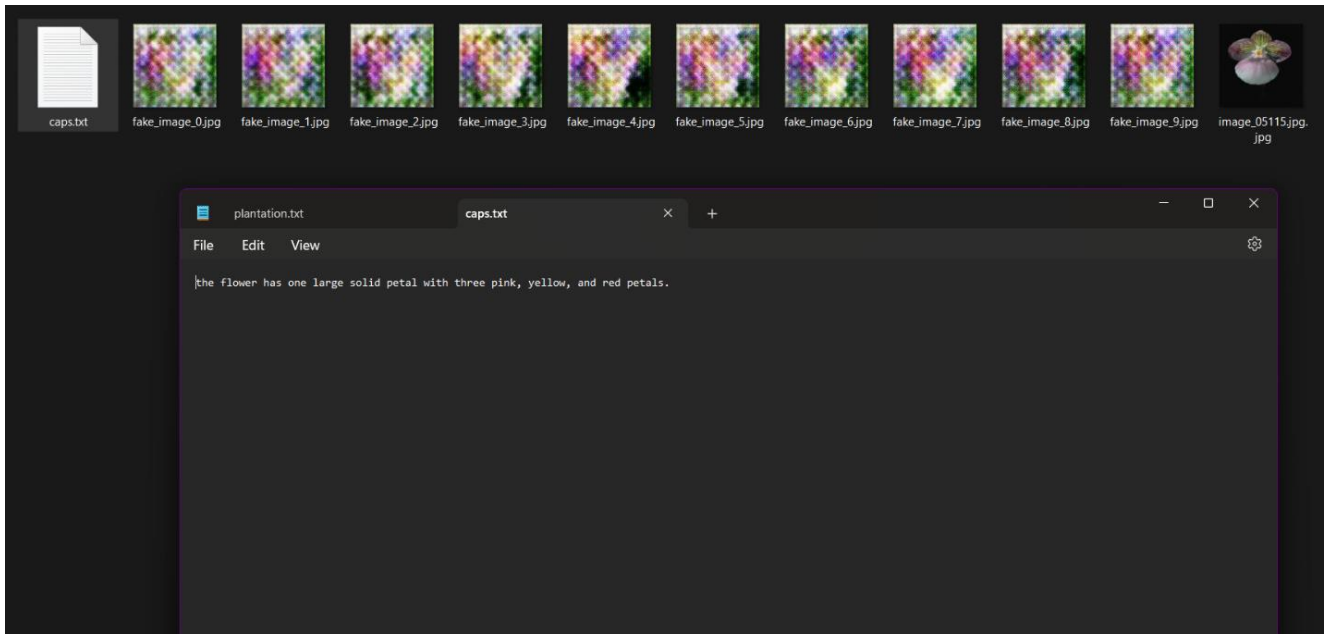


Fig 15: test result 1 Input text- flower has one large solid petal with three pink, yellow, and red petals.

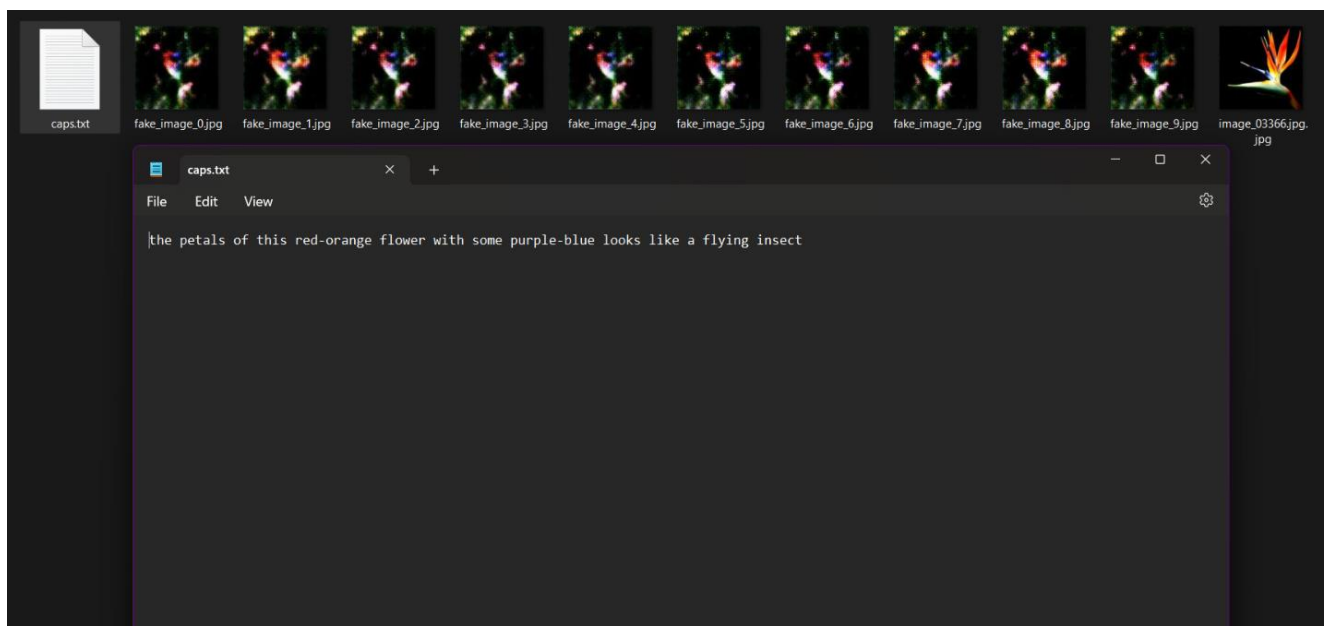


Fig 16: test result 2 Input text- petals of red-orange flower with some purple-blue like a flying insect.

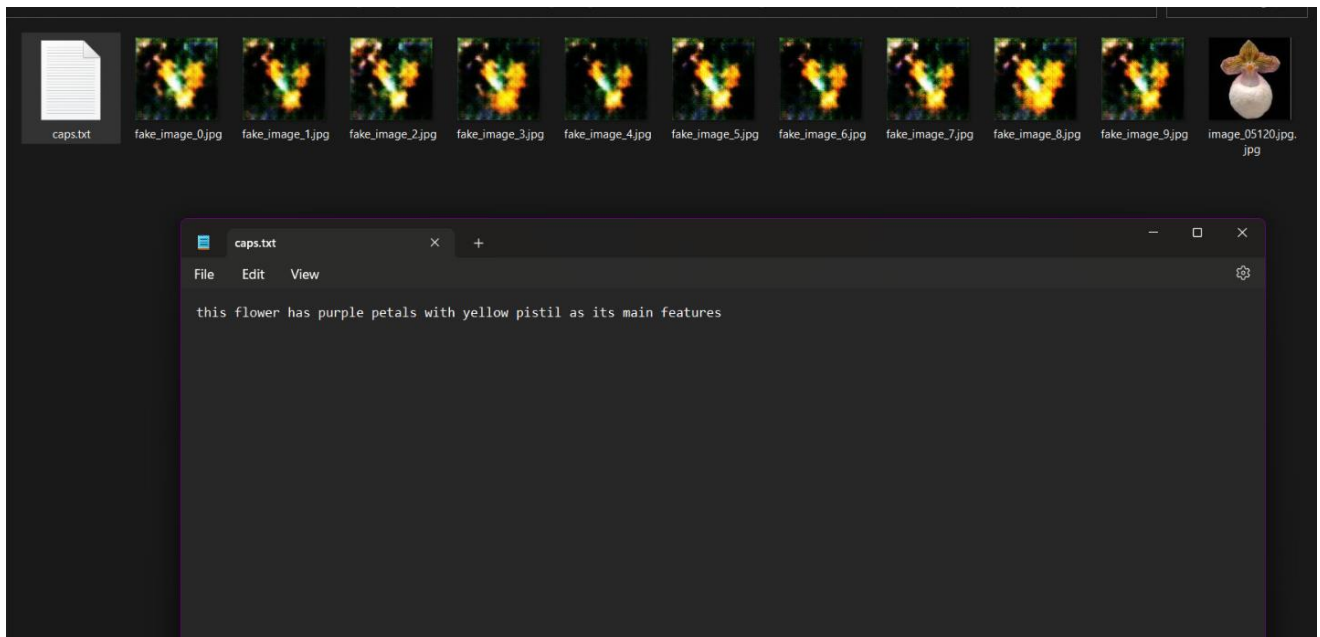


Fig 17: test result 3 Input text- flower with purple petals with yellow pistil.

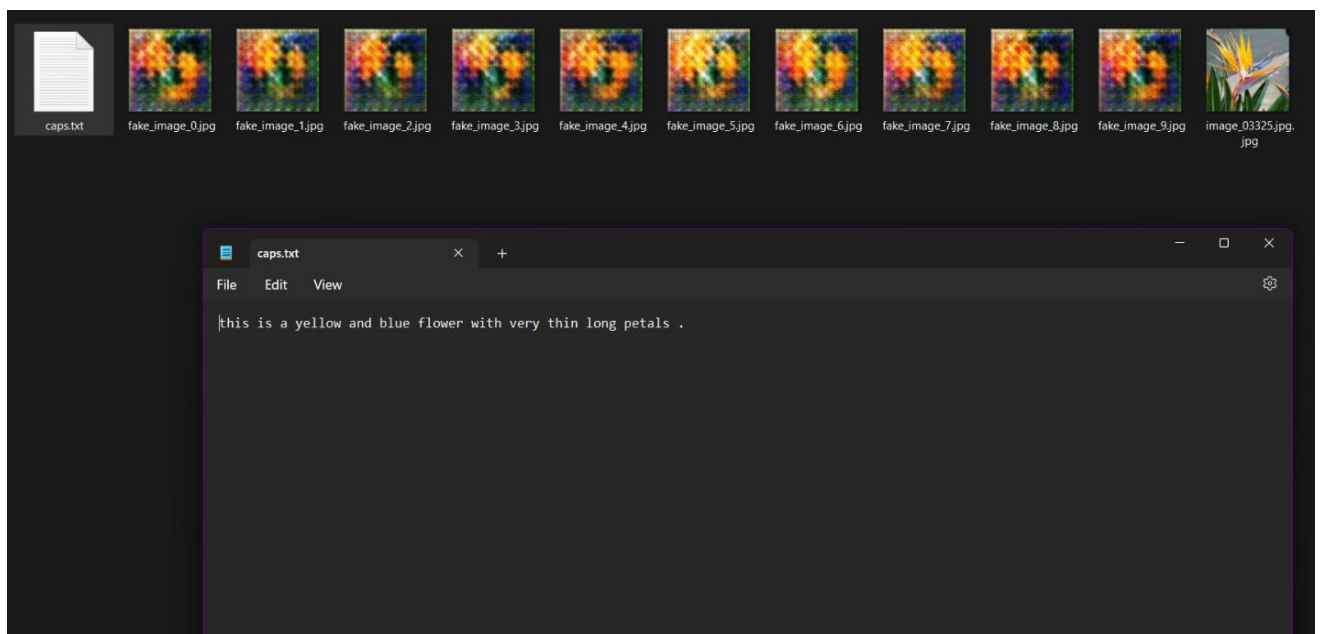


Fig 18: test result 4 Input text- a yellow and blue flower with very thin long petals.

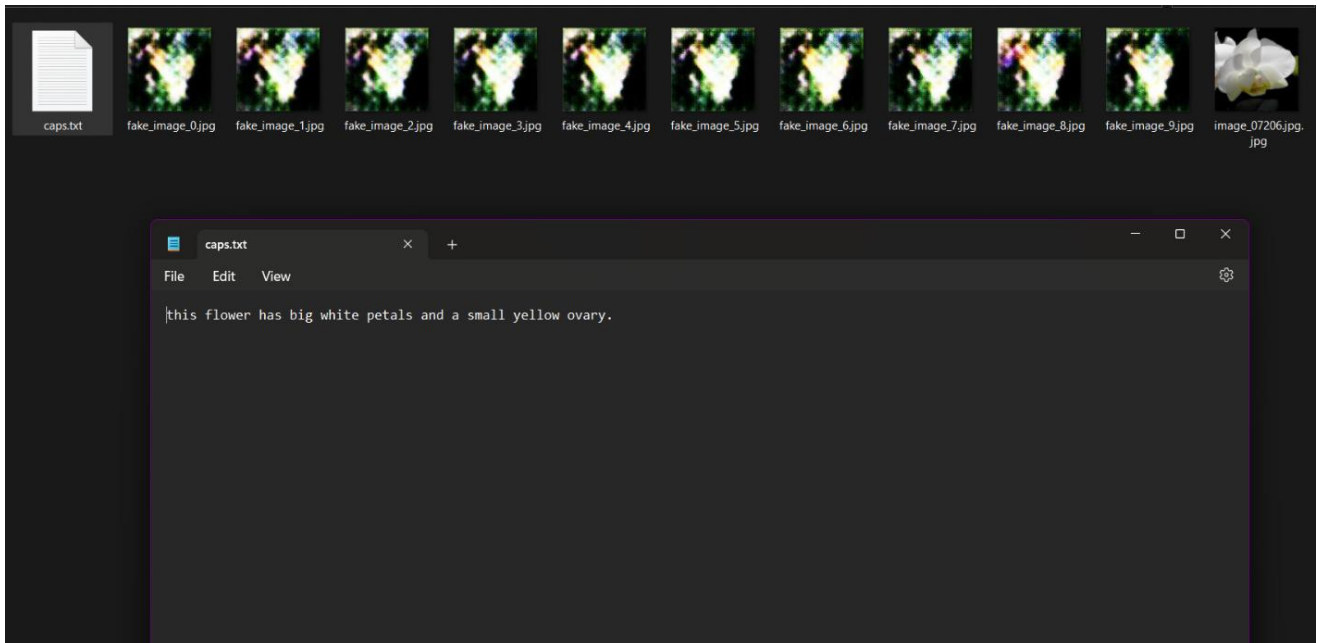


Fig 19: test result 5 Input text- *flower with big white petals and a small yellow ovary.*

One of the main strengths of TAC-GAN is its ability to generate high-quality and diverse images from textual descriptions. But our evaluation results demonstrated cannot perform well models in terms of Output and Inception Score due to lack computational resources and memory, making it difficult to train and run. Even then this is a significant improvement over previous GAN models, which often struggle to generate images that match the input descriptions.

There are also many limitations of the study that should be addressed. One of it is the evaluation metrics used in our study which may not fully capture the quality and diversity of the generated images. As Inception Score does not take into account the diversity of the generated images. Therefore, it is important to use multiple evaluation metrics and to conduct thorough visual inspection of the generated images.

In conclusion, our study demonstrates the effectiveness of TAC-GAN in generating diverse images from textual descriptions. However, much more training and testing is needed to address the limitations of the study and to fully explore the potential of TAC-GAN.

7. CONCLUSION

In conclusion, the field of text to image synthesis has seen significant progress in recent years with the emergence of Generative Adversarial Networks (GANs). One of the latest and most promising models is TAC-GAN, which has been shown to generate high-quality images that are almost indistinguishable from real images. The TAC-GAN model combines the strengths of text-to-image and GAN architectures to create a powerful framework for image generation.

The TAC-GAN model uses a text encoder to transform textual descriptions into feature vectors, which are then used by the generator to create realistic images. The discriminator is trained to distinguish between real and generated images, and the generator is optimized to produce images that fool the discriminator. By using an attention mechanism, the TAC-GAN model is able to focus on relevant parts of the text when generating images, resulting in more coherent and accurate image synthesis.

Despite its successes, there are still challenges to be addressed in the field of text to image synthesis. The generation of high-resolution images remains a difficult task, and the ability to generate images with fine-grained details and textures is still an active area of research. Additionally, the TAC-GAN model relies on a large dataset for training, which can be a limitation in some applications.

Overall, the development of TAC-GAN has opened up exciting possibilities for the generation of realistic images from textual descriptions. The potential applications for this technology are vast, from creating realistic scenes in video games to generating photorealistic images for marketing and advertising. With continued research and development, we can expect further advances in the field of text to image synthesis, paving the way for more sophisticated and powerful image generation techniques in the future.

7.1 LIMITATIONS FACED

One of the main limitations faced during the implementation of GAN for text to image synthesis on a local machine is the issue of space and time limitations. The TAC-GAN model requires a large amount of computational resources and memory, making it difficult to train and run on a local machine. Due to the large size of the model, it can take several hours or even days to train the model on a standard machine with limited resources. Additionally, the amount of memory required to store the model parameters and intermediate computations during training can quickly exceed the available memory on a local machine.

Furthermore, the training data required for our GAN model needs to be sufficiently large and diverse to enable the model to generalize well to new examples. This means that obtaining and preprocessing the training data can be a time-consuming and resource-intensive task, particularly when working with large datasets.

To overcome these limitations, researchers often resort to using powerful cloud computing services or high-performance computing clusters to train and run GAN models. Alternatively, smaller versions of the model can be trained on subsets of the data to reduce the computational and memory requirements. However, this can result in a decrease in the quality of the generated images.

In conclusion, while the TAC-GAN model shows great promise for text to image synthesis, it requires significant computational resources and memory, making it challenging to implement on a local machine with limited resources. Further research is needed to develop more efficient and scalable versions of the model that can be trained and run on standard machines without sacrificing image quality.

7.2 FUTURE SCOPE OF THE PROJECT

The development of GAN for text to image synthesis has opened up new areas of investigation and research in the field of generative models and computer vision. Some of the areas that could be explored in the future include:

1. Fine-grained control over image synthesis: GAN provides a promising approach to generating high-quality images from text descriptions.

However, there is still room for improvement in terms of fine-grained control over the image synthesis process. Researchers could investigate techniques to enable users to specify more detailed attributes of the generated image, such as the position, orientation, and scale of objects in the scene.

2. **Multimodal image synthesis:** Another area of investigation could be the synthesis of multimodal images, where the generated image includes multiple objects or scenes that are described in the input text. This would require the development of new techniques for combining multiple input modalities, such as text and images, to generate a single output image.
3. **Generalization to unseen domains:** While TAC-GAN has shown impressive results in generating high-quality images from text descriptions, there is still a challenge in generalizing the model to unseen domains. For example, the model may struggle to generate realistic images of objects or scenes that are not present in the training data. Researchers could explore techniques to improve the generalization capabilities of GAN, such as transfer learning or domain adaptation.
4. **Evaluation and benchmarking:** As with any machine learning model, it is important to evaluate the performance of GAN on standardized benchmarks and datasets. Researchers could develop new evaluation metrics and benchmarks specifically designed for text to image synthesis tasks to enable fair comparison of different models and approaches.

Overall, the development of GAN has opened up exciting new areas of investigation in the field of generative models and computer vision, and there is significant potential for further research in this area.

During the course of the text to image synthesis project using GAN, there were some parts of the work that were not completed due to time constraints and/or problems encountered. These include:

Dataset preparation: The first and foremost challenge was to prepare a suitable dataset for training the TAC-GAN model. While there are several publicly available datasets for image synthesis, finding a high-quality dataset that includes corresponding text descriptions proved to be difficult. Due to time constraints, the researchers had to settle for a smaller dataset, which may have limited the quality of the generated images.

Hyperparameter tuning: Another significant challenge was the time-consuming process of hyperparameter tuning. There are many hyperparameters to consider when training a GAN model, and finding the optimal settings can

be a time-consuming and computationally expensive task. Due to time limitations, the researchers may not have been able to explore the full range of hyperparameters to find the best settings for the TAC-GAN model.

Computational resources: Training a deep learning model like TAC-GAN requires significant computational resources, including GPUs and memory. Due to the limited resources available on the researchers' local machines, the size of the model and the training time had to be limited, which may have affected the quality of the generated images.

Lack of real-world testing: While our GAN model was able to generate high-quality images from text descriptions in the training data, there was limited testing of the model in real-world scenarios. In particular, it was not possible to test the model on a large and diverse set of text descriptions to determine its generalization capabilities.

In summary, due to time constraints and technical limitations, some aspects of the text to image synthesis project using GAN were not completed or fully explored, like in the earliest stage of the project we were implementing Stack-GAN of GAN architecture for text to image synthesis but failed due to technical resource limitations. These limitations provide opportunities for future research to improve the quality of the generated images and to explore the model's generalization capabilities in real-world scenarios.

REFERENCES

- [1] Reed, Scott, Zeynep Akata, Xincheng Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. "Generative adversarial text to image synthesis." In International conference on machine learning, pp. 1060-1069. PMLR, 2016.
- [2] Bodnar, Cristian. "Text to image synthesis using generative adversarial networks." arXiv preprint arXiv:1805.00676 (2018).
- [3] Dash, Ayushman, John Cristian Borges Gamboa, Sheraz Ahmed, Marcus Liwicki, and Muhammad Zeshan Afzal. "Tac-gan-text conditioned auxiliary classifier generative adversarial network." arXiv preprint arXiv:1703.06412 (2017).
- [4] Cain, Ryan, Gabriel Kralik, and Campbell Munson. "Photo-Realistic Image Synthesis from Text Descriptions." (2020).
- [5] Huang, He, Philip S. Yu, and Changhu Wang. "An introduction to image synthesis with generative adversarial nets." arXiv preprint arXiv:1803.04469 (2018).
- [6] Frolov, Stanislav, Tobias Hinz, Federico Raue, Jörn Hees, and Andreas Dengel. "Adversarial text-to-image synthesis: A review." *Neural Networks* 144 (2021): 187-209.
- [7] Kiros, Ryan, Yukun Zhu, Russ R. Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. "Skip-thought vectors." *Advances in neural information processing systems* 28 (2015).
- [8] Odena, Augustus, Christopher Olah, and Jonathon Shlens. "Conditional image synthesis with auxiliary classifier gans." In International conference on machine learning, pp. 2642-2651. PMLR, 2017.