EEE 443/543 Neural Networks
Final Project Report

Text to Image Generation

İhsan Doğramacı Bilkent University
Department of Electrical and Electronics Engineering

Members of Group 8:

Emir Ergin - EEE - 22102522

Muhammed Enes Adıgüzel - EEE - 21902154

Hüseyin Sefa Coşkun - EEE- 22103265

Eray Gündoğdu - EEE - 22101962

Mustafa Cankan Balcı - EEE - 22101761

Date: 10.01.2025

Table of Contents

Abstract

This project aims to investigate advanced and comprehensive methodologies in text-to-image synthesis, making use of state-of-the-art models and a custom architecture to translate textual descriptions into high quality visual representations. The dataset used for the model training, testing and validation is the COCO 2017 dataset. Three pre-trained models were used, which are making use of diffusion processes, latent representations, and cross attention mechanisms, while the custom model implements a merging approach on two models, which are OpenAI's CLIP and Generative Adversarial Networks, named CLIP-GAN.

1. Introduction

Text-to-image generation has emerged as a transformative domain within neural networks, showcasing the ability of machine learning models to translate natural language descriptions into visually coherent images. This project aims to explore the application of advanced text-to-image models, employing state-of-the-art architectures such as GLIDE, Stable Diffusion, and PixArt, alongside the development of a custom CLIP-GAN model. These models are comprehensive of the recent developments of neural networks, in which they explore various different approaches, such as diffusion processes, latent representatiins and generative adversarial networks (GANs). Besides analysing the properties of the pre-trained models, this project further highlights the contributions of the custom model. The project compares the efficacy of the models based on both qualitative and quantitative metrics, such as Fréchet Inception Distance (FID), Inception Score (IS), and CLIP Score.

2. Methods

The COCO (Common Objects in Context) 2017 dataset, originally introduced by Lin et al. [1] and further documented on the official website [2], is a large-scale image collection spanning a wide range of everyday scenes. It includes detailed annotations for multiple computer vision tasks, such as object detection, segmentation, and especially image captioning. The 2017 release contains approximately 118,000 training images, 5,000 validation images, and 41,000 test images. Each training image is paired with several human-written captions describing its content, making the dataset particularly well-suited for text-to-image generation research.

Although COCO provides a dedicated test set, it does not include official captions for those images, which prevents direct evaluation of text-to-image models on that subset. As a result, we use only the training portion of COCO 2017 for model development. To preserve caption annotations during every phase of experimentation, we split the official training subset into three parts: 80% for training, 10% for validation, and 10% for testing. This approach ensures that each image in our custom splits retains its ground-truth captions, allowing for thorough model training and subsequent performance evaluation.

Overall, COCO 2017 serves as an extensive and consistently annotated resource that aligns well with the requirements of text-conditioned image synthesis.

Through the use of COCO 2017, we have implemented 3 pre-trained models of different architectures and complexities, and a custom model designed from scratch which combines the CLIP of OpenAI and the GAN architecture, which is labeled as CLIP-GAN. The 3 pre-trained architectures used are GLIDE, Stable Diffusion and PixArt respectively.

2.1. GLIDE

In this section, we describe how we implemented GLIDE (Guided Language-to-Image Diffusion for Generation and Editing), a diffusion-based text-to-image model presented by Nichol et al. (2022). GLIDE is a key component in our text-to-image generation workflow, using iterative denoising steps to transform natural language descriptions into high-fidelity, semantically consistent images [3].

2.1.1. Architecture

GLIDE relies on a diffusion process to generate images by gradually refining noisy samples until they converge to clean, high-quality images aligned with textual prompts:

2.1.1.1. Forward Diffusion Process

In the forward (noising) direction, the model starts with a clean image x_0. Over T time steps, it progressively corrupts this image through a Markov chain, producing increasingly noisy versions $x_1, x_2, ..., x_T$. At each step t, Gaussian noise is added according to a specified variance schedule α_T. Formally, we can write:

$$q(x_t|x_{t-1}) = N(x_t; \sqrt{\alpha_t}x_{t-1}, (1 - \alpha_t)I)$$

As t increases, x_t becomes progressively more corrupted, eventually approaching an isotropic Gaussian distribution.

2.1.1.2. Reverse Denoising Process

The core idea of GLIDE is to learn the reverse process that starts from pure noise x_T and iteratively removes noise, reconstructing a clean image x_0. The reverse conditional distribution is parameterized by a neural network:

$$p_\theta(x_{t-1}|x_t, c)$$

where c represents the conditioning information derived from a text encoder. At each step, the network predicts the denoised image, guiding the sample from x_t back to x_{t-1}.

2.1.1.3. UNet-Based Denoising BackBone

Following the paper by Nichol et al., 2022, GLIDE uses a UNet-style convolutional architecture to perform the denoising. The network includes the following:

- **Downsampling Blocks:** Repeated 2D convolutions systematically reduce the spatial resolution of the noisy input, allowing the model to capture increasingly coarse-grained features.
- **Middle Blocks:** Residual blocks with attention mechanisms that integrate global context, including the text embedding.
- **Upsampling Blocks:** In a mirrored process, these blocks reconstruct higher-resolution spatial details, ultimately producing the refined image output.

2.1.1.4. Text Conditioning

To integrate textual information, the model applies cross-attention or feature-wise transformations. A transformer-based text encoder extracts embeddings from the provided captions or prompts, which are then introduced into the UNet through specialized attention layers. This way, each denoising step is informed by semantic cues derived from the text, ensuring that the generated image remains closely aligned with the given description.

2.1.2. Classifier-Free Guidance

A key innovation of GLIDE, as described by Nichol et al., 2022, is its classifier-free guidance mechanism, which removes the need for a separately trained classifier to align images with textual prompts. During training, the model is sometimes given a real text prompt (conditional mode) and sometimes given a "null" prompt (unconditional mode). This dual-mode training strategy enables the model to learn both how to integrate text constraints and how to generate images without explicit textual guidance.

At inference time, the model produces two noise estimates: one unconditional $\epsilon(x_t)$ and one conditional on the text prompt $\epsilon(x_t, c)$, where c represents the encoded caption. These two estimates are then combined as

$$\hat{\epsilon}(x_t, c) = (1 + \omega) \cdot \epsilon(x_t, c) - \omega \cdot \epsilon(x_t)$$

with ω denoting the guidance scale. Larger values of ω force the generated images to adhere more strongly to the textual description but may reduce overall diversity. Conversely, lower values of ω strike a balance between prompt fidelity and variability in generated samples, thus allowing for more creative or unexpected outputs.

2.1.3. Training Procedure

The GLIDE model is trained by following the standard diffusion-based framework, with several noteworthy details regarding data preprocessing, loss computation, and hyperparameter selection:

2.1.3.1. Data Preprocessing

All images are resized and/or cropped to a resolution of 64×64. Textual prompts from the COCO 2017 captions are tokenized and embedded using a pre-trained language model (e.g., CLIP text encoder). These image-text pairs form the core training samples.

2.1.3.2. Loss Function

The model is optimized to predict the Gaussian noise added at each diffusion step. Let $\epsilon_\theta(x_t, t, c)$ denote the model's noise prediction, where x_t is the noisy image at step tt, and cc represents the text embedding. The mean-squared error (MSE) between the predicted noise and the true noise is minimized:

$$L(\theta) = E_{t,x_0,\epsilon}[||\epsilon - \epsilon_\theta(x_t, t, c)||^2]$$

This objective function drives the reverse denoising process so that the final outputs approximate the original, clean images in a text-conditioned manner.

2.1.3.3. Hyperparameters

Typical training configurations include a diffusion step count $T\epsilon\{100, 250, 1000\}$, where higher values of T generally improve image quality at the expense of increased computational cost. The AdamW optimizer is commonly chosen with a learning rate of around $1 * 10^{-4}$, decaying gradually during training. A batch size of 64 often provides a reasonable trade-off between computational feasibility and training stability.

2.1.4. Implementation Details

The open-source implementation of GLIDE is based on PyTorch, leveraging efficient operations and GPU acceleration to handle large-scale diffusion. Mixed-precision (float16) training is typically adopted to reduce GPU memory usage and training time without substantially degrading model performance.

At inference time, the model starts from a random Gaussian sample at the final diffusion timestep T. The reverse diffusion process is then applied iteratively, gradually denoising the sample at each step until the final image is obtained. For text alignment, a guidance scale ω is tuned to balance semantic fidelity against sample diversity. A higher ω value enforces stronger adherence to textual prompts, whereas a lower ω fosters more diverse image outputs [3].

2.2. Stable Diffusion

In this section, we explore Stable Diffusion, a cutting-edge text-to-image generation model powered by a latent diffusion process. The core ideas behind it trace back to Latent Diffusion Models (LDMs), as introduced by Rombach et al. in 2022 [4]. The following figure offers a diagrammatic overview of the open-source version of the Stable Diffusion model [5].

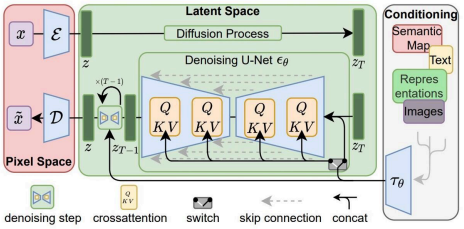


Figure 1: Diagram of Stable Diffusion [7]

2.2.1 Latent Diffusion Framework

Unlike conventional diffusion models that operate on raw pixel values, Latent Diffusion Models first learn a latent representation of images. This approach leverages an autoencoder to map an image x to a latent variable z. By working in this domain, the diffusion process is performed on z rather than x. Once the latent variable is sufficiently denoised, a decoder reconstructs a high-resolution image from the clean latent.

2.2.1.1. Encoder-Decoder Autoencoder

- **Encoder** E**:** Maps the original image $x_0 \epsilon R^{H\times W\times 3}$ to latent embedding $z_0 = E(x_0)$
- **Decoder** D**:** Reconstructs the image from the latent embedding $\widehat{x_0} = D(z_0)$.
- The latent dimensionality is considerably lower than the pixel space, making diffusion steps in the latent space more efficient.

2.2.1.2. Forward Diffusion in Latent Space

The forward process corrupts z_0 by gradually adding Gaussian noise over T time steps. A typical variance scheduling scheme $\{\beta_t\}_{t=1}^T$ is used. $(a_t = 1 - \beta_t)$ For a single step following computation is carried, then this computation is carried over T steps.

$$q(z_t|z_{t-1}) = N(z_t; \sqrt{\alpha_t}z_{t-1}, (1 - \alpha_t)I)$$

2.2.1.3. Reverse Diffusion (Denosing)

A UNet-based network $\epsilon_\theta(z_t, t, c)$ is trained to predict the added noise ϵ at each step t, given both the noisy latent z_t and the text condition c.

By iteratively refining noise estimates in latent space, the model recovers a noise-free latent z_0. The decoder subsequently reconstructs the image in pixel space.

2.2.2. Model Architecture

The Stable Diffusion pipeline can be broken down into three main components:

2.2.2.1. Text Encoder

A pre-trained transformer, such as the CLIP Text Encoder, extracts embeddings from the input prompt. These text embeddings then guide the diffusion process through cross-attentiin mechanism within the UNet, ensuring that the generated images align with the given prompt.

2.2.2.2. Unet Denoising Network

The UNet operates on latent space representations, employing a series of downsampling and upsampling blocks interspersed with attention layers. This structure enables both local and global context aggregation, while skip connections preserve important structural details throughout the denoising process.

2.2.2.3. VAE Decoder

The decoder converts the denosied latent z_0 back into the final high-resolution image and is typically trained alongside the diffusion model to ensure consistent, perceptually meaningful latent encodings.

2.2.3. Derivation and Key Insights

The derivation of Latent Diffusion Models by Rombach et al. (2022) builds upon the standard diffusion formulation but moves the entire process into a learned latent space. Let $z_0 = E(x_0)$ be the latent representation of the original image. The forward-noising distribution at timestep t is given by:

$$q(z_t|z_0) = N(z_t; \sqrt{\alpha}_t z_{t-1}, (1 - \bar{\alpha}_t)I).$$

The model then learns a parameterized reverse distribution:

$$p_\theta(z_{t-1}|z_t, c) \approx q(z_{t-1}|z_t, z_0)$$

by predicting the noise ϵ that was added at each step. The training objective becomes a simple mean-squared error (MSE) between the predicted noise $\epsilon_\theta(z_t, t, c)$ and actual noise ϵ.

A key breakthrough is that compressing images into a smaller latent space significantly cuts down the dimensionality of the diffusion process, yet still preserves perceptual quality—thanks to the autoencoder's powerful decoding. Additionally, text conditioning is incorporated through cross-attention in the UNet, making the generated images highly responsive to the input prompt.

2.2.4. Inference Procedure

During inference, a random latent z_T is sampled from a Gaussian distribution. The model then applies its learned reverse diffusion steps:

$$z_{t-1} = z_t - (predicted\ noise)$$

to iteratively refine the noise until reaching z_0. This final latent is then decoded into an image by the VAE. Similar to classifier-free guidance methods, a guidance scale can be used to balance how strictly the model follows the prompt c versus how creatively it explores different outputs [4,5].

2.3. PixArt

PixArt refines latent-space diffusion with enhanced cross-attention modules and classifier-free guidance, resulting in high-resolution, semantically aligned image generation even for complex prompts. By employing a carefully tuned autoencoder, it preserves fidelity and ensures that the final outputs accurately reflect the text prompts [6].

2.3.1. Model Architecture

PixArt encodes images into a lower-dimensional latent space via an autoencoder. A UNet-based diffusion network, equipped with cross-attention, then denoises these latents over T timesteps. Finally, a decoder reconstructs the high-resolution image from the denoised latent.

2.3.2. Classifier-Free Guidance

PixArt adopts a classifier-free guidance approach to handle both conditional (prompted) and unconditional (prompt-free) generation. During training, the text prompt is randomly dropped with a small probability,

allowing the model to learn two noise estimators:

$$\epsilon_{cond}(z_t, c),\ \epsilon_{uncond}(z_t)$$

where z_t is the noisy latent representation at timestep t, and c s the text embedding. At inference, these estimates are interpolated using a guidance scale ω:

$$\hat{\epsilon}(z_t, c) = (1 + \omega)\epsilon_{cond}(z_t, c) - \omega\epsilon_{uncond}(z_t)$$

2.3.4. Training Procedure and Loss Objective

During training, images are scaled to a chosen resolution and embedded into latent space via the autoencoder encoder. Let z_0 denote the clean latent and z_t the noisy latent at diffusion step t. PixArt is optimized via a mean-squared error (MSE) loss:

$$L(\theta) = E_{t,z_0,\epsilon}[||\epsilon - \epsilon_\theta(z_t, t, c)||^2]$$

where ϵ is the noise added at each forward diffusion step and $\epsilon_\theta(\cdot)$ is PixArts's predicted noise. Typical hyperparameters include 200–1000 diffusion steps, the AdamW optimizer at a learning rate of 1e-4, and mixed-precision (float16) training to reduce computational overhead.

2.4 CLIP-GAN

We have also included the methodology of CLIP-GAN (Contrastive Language-Image Pre-training - Generative Adversarial Networks) in our project for the text-to-image generation task. This methodology is a combination of the GAN [8] by Goodfellow et. al (2014), and OpenAI's CLIP [9]. CLIP-GAN models are especially powerful in overcoming the issue of the gap between visual and linguistic modalities. In specifics, CLIP's text encoder provides textual embeddings that guide the image generation process, while its image encoder evaluates how accurately the generated images correspond to the textual inputs.

2.4.1. Architecture

The architecture of a CLIP-GAN is comprised of two main elements, which are;

2.4.1.1. Generator

The generator is a deep convolutional network that takes as input the random noise vector, and a text embedding, to generate a high resolution image conditioned on the text input

2.4.1.2. Discriminator

Discriminator is another convolutional network that evaluates the authenticity of the image as well as comparing its correspondence with the textual description provided to the generator.

2.4.2. Text Embedding Projection

In the input text embedding part, the captions are tokenized and passed through the CLIP's text encoder to generate a high-dimensional vector called e_{CLIP} where the projection layer is mapped to the required space by the CLIP embedding through linear transformation, in order to adjust the scale, dimensionality and the feature weights. The equation is:

$$e_{text} = W_{proj} \bullet e_{CLIP} + b_{proj}$$

where W_{proj} is the learnable weight matrix, b_{proj} is the learnable bias vector and e_{text} is the projected text embedding.

2.4.3. Generator Forward Pass

A random noise vector and a projected text embedding is taken and transformed into a high resolution image that aligns semantically with the text. A latent noise vector z is used as the noise component in the forward pass function. The overall generator forward pass equations are as provided below:

$$z_{input} = Concat(z, e_{text})$$

$$x_0 = ReLU(W_{init} \bullet z_{input} + b_{init})$$

$$x_{i+1} = ReLU\left(ConvTranspose2D\left(x_i, W_i, b_i\right)\right)$$

Through the implementation of the described process, the images are generated.

2.4.4. Discriminator Forward Pass

The discriminator is assigned the mission of evaluating an image along with its text embedding to determine whether the image is real or generated. The essential proceedings of the discriminator is the combination of image features and projected text embeddings for classification. The input image x of the discriminator could either be a real image

sampled from the dataset or a fake image generated by the generator module. The derivations used for the discriminator are as follows:

$$f_{img} = Flatten\left(Conv2D(x, W_{conv}, b_{conv}\right)$$

$$z_{combined} = Concat\left(f_{img}, e_{text}\right)$$

$$y = \sigma\left(W_{final} \bullet z_{combined} + b_{final}\right)$$

where f_{img} stands for the extracted image features and the σ is the sigmoid activation function for binary classification. Following this logic, the output y is a probability value, which indicates that the discriminator predicted the image to be real when the value is close to 1, whereas a value close to 0 would indicate that the image is labeled as fake, or it does not align with the text.

2.4.5. Adversarial Loss

Adversarial loss is an essential part of a GAN, which is used for the training of both the generator and the discriminator. It involves the binary cross-entropy loss in its computations which quantifies the difference between predicted probabilities and true labels in the form of a binary classification problem. Considering $D(\cdot)$ and $G(\cdot)$ as discriminator and generator outputs respectively, the losses can be computed as:

$$L_D = -\ E_{X\sim p_{real}}[logD(X)] - E_{Z\sim p_{noise}}\left[log\left(1 - D\left(G\left(z, e_{text}\right)\right)\right)\right]$$

$$L_G = -E_{Z\sim p_{noise}}\left[logD\left(G\left(x, e_{text}\right)\right)\right]$$

2.4.6. CLIP Similarity Loss

The purpose of the clip similarity loss computation is to ensure the semantic consistency between the generated image and the input text description. Based on the computed loss, the generator is penalized if the generated image embedding is not similar to the text embedding of the input description. The similarity is computed using cosine similarity defined as:

$$CosSim\left(e_{image}, e_{text}\right) = \frac{e_{image} \bullet e_{text}}{||e_{image}||\,||e_{text}||}$$

Therefore, CLIP similarity loss is computed as:

$$L_{CLIP} = -CosSim\left(e_{image}, e_{text}\right)$$

2.4.7. Training Overview

The training of the CLIP-GAN model is completed over the COCO 2017 dataset with the aforementioned details regarding the data splitting parameters and the validation images. Initially, all the images in the training data set were preprocessed to match the input requirements of the CLIP architecture. The images were resized to 224 x 224 pixels and normalized based on their mean and standard deviation values. Texts are tokenized using CLIP's tokenizer, converting each text string into a sequence of tokens compatible with CLIP's text encoder. Since we know intuitively that the discriminator will tend to learn faster than the generator, as it is solving a simpler classification problem, without proper tuning, the discriminator will overpower the generator. Due to this reason, the training is done with a Two-Time Scale Update rule (TTUR). TTUR is a technique used in training GANs to stabilize adversarial training by employing different learning rates for the generator and discriminator [10]. This prevents the imbalance in GAN training, and brings it to a state of stability through avoiding vanishing gradients and mode collapses [10].

3. Results

This section presents the outcomes of our text-to-image generation experiments, conducted on the COCO 2017 dataset using four distinct models. Three of these models (GLIDE, Stable Diffusion, and PixArt) are pre-trained neural architectures sourced from publicly available implementations. The fourth model is a custom, in-house design trained from scratch (CLIP-GAN). In the following subsections, both qualitative (visual comparisons) and quantitative (e.g., Fréchet Inception Distance, Inception Score, CLIP Score) evaluations are provided to compare how effectively each model captures semantic details and produces realistic images aligned with textual prompts.

3.1 GLIDE

GLIDE's output quality was evaluated on the COCO 2017 dataset using three commonly adopted metrics in generative modeling: Fréchet Inception Distance (FID), Inception Score (IS), and CLIP Score. Table 1 summarizes the numerical performance of GLIDE. A lower FID indicates a closer match between the model's generated image distribution and the real data distribution, while

a higher IS suggests greater diversity and recognizability of generated samples. The CLIP Score measures the semantic alignment between generated images and their corresponding text prompts. All metrics consistently show that GLIDE is capable of producing visually coherent and textually aligned images.

Scores	GLIDE
FID	216
IS	1.91 ± 0.08
CLIP	0.3156

Below Figures illustrate sample outputs generated by GLIDE for three distinct textual prompts: *"a cow is grazing in the grass near tall reeds [1],"* *"several bundles of bananas [2],"* and *"a street sign with graffiti [3]."* These examples demonstrate GLIDE's capability to interpret and render diverse visual concepts from natural language descriptions, highlighting the model's ability to capture both background elements (e.g., reeds, grass, urban scenery) and specific objects (cows, bananas, signage) accurately.

[1]

[2]

[3]

GLIDE was trained on large-scale text–image data using a UNet-based diffusion model conditioned on text embeddings, employing a noise schedule of up to 1000 steps. The model used AdamW at a learning rate of 1e-4 with cosine decay, a typical batch size between 32 and 256, and classifier-free guidance by randomly dropping text prompts to handle both conditional and unconditional paths.

3.2 Stable Diffusion

Stable Diffusion, which operates in a learned latent space rather than pixel space, yielded competitive performance on the COCO 2017 dataset. The following subsections present the key metrics such as Fréchet Inception Distance (FID), Inception Score (IS), and CLIP Score along with sample outputs demonstrating the model's ability to produce high-quality images that closely align with the given textual prompts.

Based on the evaluation of the generated outputs, the following table was compiled to present the quantitative performance metrics of Stable Diffusion on the COCO 2017 dataset.

Scores	Stable Diffusion
FID	192.8
IS	1.75 ± 0.21
CLIP	0.3229

When creating this table, the images used to derive the reported data can be examined in detail in Appendix A.

Subsequently, various inputs were fed to the model, producing the images shown below. The same captions were retrieved from the COCO 2017 dataset to obtain corresponding ground-truth images, allowing for a direct comparison of output quality.

[1]


[2]

Specifically, the figures above illustrate the model's results for the prompts *"a kid is in the air on a skateboard" [1]* and *"a white motorcycle parked on top of green grass." [2]* For additional sample prompts and their corresponding outputs, as well as the ground-truth images, please refer to Appendix B. (The image on the left represents the generated output, while the image on the right displays the corresponding ground truth.)

Stable Diffusion is a latent diffusion method that first encodes images into a compressed VAE latent space, applies a UNet-based diffusion model, then decodes back to the full image. Following Rombach et al. (2022), it is trained on large-scale text–image datasets with around 250–1000 diffusion steps. The model typically uses AdamW at a learning rate of 1e-4, a moderate batch size, incorporates classifier-free guidance for text conditioning, and often employs mixed precision (float16) for efficiency.

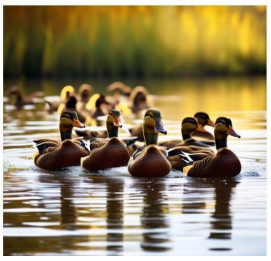
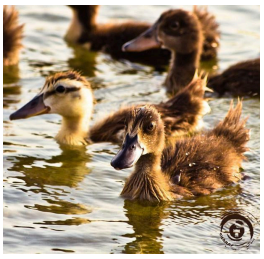
3.3 PixArt

PixArt has been the model with the highest performance in the project with the best scores obtained when compared to the other two pre-trained models. This can be seen intuitively from the quality of the images generated (in terms of similarity with ground truth), as well as the quantitative analysis provided in the below table. Besides the table, image pairs are also provided both below and in the Appendix C part of this report. The similarity between the generated image and the ground truth is clearly observable.

Scores	PixArt
FID	113.6
IS	1.76 ± 0.09
CLIP	0.3389

When creating this table, the reported data can be examined in detail in Appendix D.

Subsequently, the model was provided with various inputs to generate images, and corresponding ground-truth images with the same captions were also retrieved. One such result is illustrated in the example below, using the following caption: *"a flock of brown ducks swimming together on a lake." [1]*

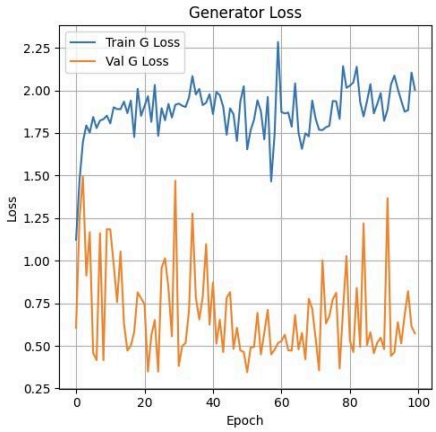
[1]

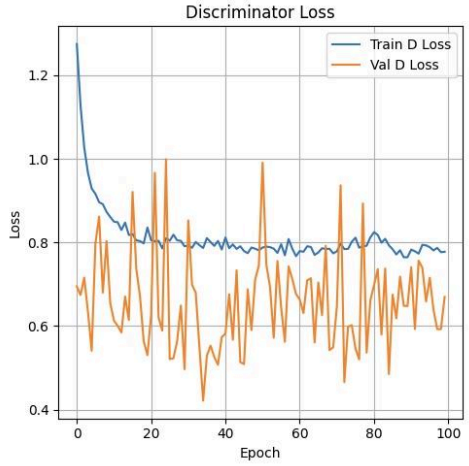
(The image on the left represents the generated output, while the image on the right displays the corresponding ground truth.)

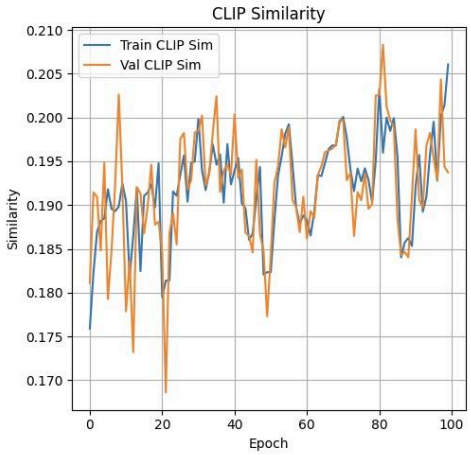
PixArt's training procedure begins by resizing images to a standardized resolution and encoding them into a latent space via an autoencoder. A forward noise schedule of 200–1000 steps is used to corrupt these latent representations, and the model is trained to denoise them using a mean-squared error (MSE) objective. The paper typically employs the AdamW optimizer at a base learning rate of around 1e-4, often with a small weight decay, and a moderate batch size (e.g., 32–128). To support classifier-free guidance, the text prompt is dropped with a certain probability during training. Mixed-precision (float16) is adopted to reduce memory overhead, and minimal data augmentations such as random flips are applied to improve robustness.

3.4 CLIP-GAN

Having used the aforementioned methodology for computation derivation and training, the following results were obtained for the CLIP-GAN model, which is trained via COCO 2017 dataset. As we have previously discussed, the adversary loss and clip similarity plots are obtained as well as 3 different generated images based on the user inputs.







[Loss & Similarity Plots for CLIP-GAN]

The images provided above are obtained by training the code with the following hyperparameters:

$$Latent\ Dimension\ =\ 100,\ Batch\ Size\ =\ 32$$

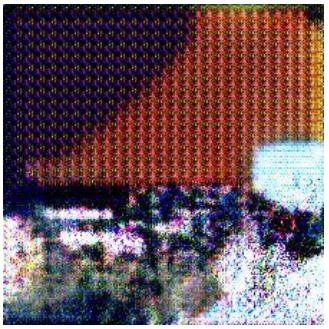
$$Epoch\ Size\ =\ 100,\ Image\ Size\ =\ 224$$

$$\eta_{generator}\ = 10^{-4}\ \&\ \eta_{discriminator}\ =\ 10^{-5}$$

The generated images with the user inputs below are provided in the following images.

[a bed above a desk]

[an old aircraft]



[black cat sitting]

The training of the model was completed by making use of Google Collab software with the Nvidia A100 GPU, which took 7 hours and 42 minutes.

4. Discussion

Various state-of-the-art machine learning algorithms have been applied to the COCO 2017 training dataset, with each model exhibiting unique advantages and disadvantages, among other characteristics. This diversity in model performance underscores the importance of selecting appropriate algorithms based on specific application requirements and dataset complexities.

Our findings align with existing studies and highlight the strengths and limitations of text-to-image generation algorithms. GLIDE achieved consistent text-to-image alignment, as observed by Nichol et al. (2022), but its high computational requirements reaffirm its known trade-off between efficiency and quality. Although the model performed well on simpler prompts, it struggled with complex scenes, suggesting that better fine-tuning can be implemented.

Similarly, Stable Diffusion, as noted by Rombach et al. (2022), performed effectively in latent space generation but occasionally simplified complex scene details. This suggests that while using a compressed latent space improves efficiency, it may reduce fine details. Enhancing the VAE decoder or using more detailed embeddings could help address this issue.

PixArt's results reflect findings from recent studies that emphasize the role of improved cross-attention mechanisms in handling high-resolution synthesis, demonstrating the impact of architectural choices on performance. However, despite performing better than GLIDE and Stable Diffusion in terms of FID and CLIP scores, its high computational requirements are non-negligible. Furthermore, its scores can increase significantly by applying better fine-tuning, which was limited in this project considering the training alone takes several days.

To further explore custom architectures, CLIP-GAN model is applied for text-to-image generation tasks. The overall approach on the implementation of the CLIP-GAN model is found to be correct during the training process. However, the model underperforms on the testing data, suggesting overfitting or instability during training. This aligns with known challenges in GAN-based models, such as mode collapse and the sensitivity of adversarial loss functions. Future iterations could focus on improved training strategies, (besides the already applied TTUR) such as dynamic learning rate adjustments or additional regularization techniques, to enhance stability and generalization.

Moreover, to enhance the text to image synthesis performance, we conducted new analysis in addition to analyses in the papers by utilizing hierarchical text captioning approach. Instead of processing captions as simple flat sequences of tokens, we structured the captions hierarchically to distinguish between global scene descriptors (e.g., "a busy city street") and specific object-level details (e.g., "pedestrians crossing"). This hierarchical representation was incorporated into the training process by adding an additional attention mechanism that assigns weights differently to global and local descriptors during image generation. This hierarchical weighting mechanism uncovered a natural progression in the neural network's attention, moving from general to specific details during the generation process. This finding suggests that text-to-image models can benefit from explicit structuring of input captions, which mirrors how humans may describe and visualize scenes.

Further analysis is conducted on new datasets which are the CUB (Caltech-UCSD Birds 200) dataset and the real-life StreetScenes-2025 dataset alongside COCO 2017. The CUB dataset is not considered as a real-life dataset, however it is crucial to understand how models perform in a domain specific dataset whereas StreetScenes-2025 includes complex descriptions. In CUB, we recovered a structure that emphasizes fine-grained, domain-specific details such as species-specific characteristics and singular focus on the primary object (the bird), with minimal background influence. In contrast, the StreetScenes dataset revealed a more dynamic structure involving complex relationships between multiple objects and temporal or spatial interactions. COCO 2017, balances global scene context hence we reached to the conclusion that the models trained on COCO 2017 dataset performed relatively sufficient in describing domain-specific CUB captions, however the models should be fine-tuned in order to capture more complex relations as in StreetScenes-2025 real-life dataset.

By maintaining our research efforts and conducting further experiments, we can continuously enhance and refine these models. In the realm of text-to-image synthesis, there remains significant potential to generate more accurate and captivating visuals from written descriptions. This can be achieved by overcoming existing challenges and exploring new architectural designs and training methodologies.

Overall, the implemented methodology demonstrates significant potential for bridging the gap between textual descriptions and visual representations, thereby substantially advancing the field of text-to-image synthesis.

The implemented models that we have preferred to train and evaluate have much room to improve given that there could be many more modifications to their hyperparameters, as well as the complexity of their structure.

References

[1] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A., "Going deeper with convolutions," *arXiv preprint arXiv:1405.0312*, 2014. [Online]. Available: https://arxiv.org/abs/1405.0312.

[2] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," *ECCV 2014*, 2014. [Online]. Available: https://cocodataset.org/#home.

[3] Nichol, A., Dhariwal, P., Ramesh, A., Shyam, P., Mishkin, P., McGrew, B., Sutskever, I., & Chen, M., "GLIDE: Towards photorealistic image generation and editing with text-guided diffusion models," *arXiv preprint arXiv:2112.10741*, 2021. [Online]. Available: https://arxiv.org/abs/2112.10741.

[4] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. "High-Resolution Image Synthesis with Latent Diffusion Models," *arXiv preprint arXiv:2112.10752*, 2022. [Online]. Available: https://arxiv.org/pdf/2112.10752.

[5] hkproj, "PyTorch Stable Diffusion," GitHub repository, 2025. [Online]. Available: https://github.com/hkproj/pytorch-stable-diffusion?tab=readme-ov-file [Accessed: 10-Jan-2025].

[6] Chen, J., Ge, C., Xie, E., Wu, Y., Yao, L., Ren, X., Wang, Z., Luo, P., Lu, H., and Li, Z., "PixArt-Σ: Weak-to-Strong Training of Diffusion Transformer for 4K Text-to-Image Generation," *arXiv preprint arXiv:2412.12391*, 2024. [Online]. Available: https://arxiv.org/abs/2412.12391.

[7] Xue, B. "Stable Diffusion: A Tutorial," Harvard Scholar. [Online]. Available: https://scholar.harvard.edu/binxuw/classes/machine-learning-scratch/materials/stable-diffusion-scratch. [Accessed: Jan. 9, 2025].

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139-144, 2014. Available: https://www.researchgate.net/publication/263012109_Generative_Adversarial_Networks.

[9] OpenAI, "CLIP: Contrastive Language–Image Pretraining," OpenAI, 2021. [Online]. Available: https://openai.com/index/clip/. [Accessed: Jan. 09, 2025].

[10] F. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1706.08500.

[11] OpenAI, "GLIDE: Text-to-Image," GitHub repository, 2021. [Online]. Available: https://github.com/openai/glide-text2im [Accessed: 10-Jan-2025].

[12] PixArt-alpha, "PixArt-sigma," GitHub repository, 2025. [Online]. Available: https://github.com/PixArt-alpha/PixArt-sigma [Accessed: 10-Jan-2025].

Appendix

A. Score Outputs of the Stable Diffusion Model

```
Caption: a kid is in the air on a skateboard
FID Score (Text-to-Image): 275.7092
```

```
Caption: a couple of doughnuts that is on a plate
FID Score (Text-to-Image): 119.0567
```

```
Caption:  A white toilet sitting beneath a wood and glass cabinet.
FID Score (Text-to-Image): 124.5507
```

```
Caption:  A group of police motorcycles leading a firetruck in a parade.
FID Score (Text-to-Image): 206.2859
```

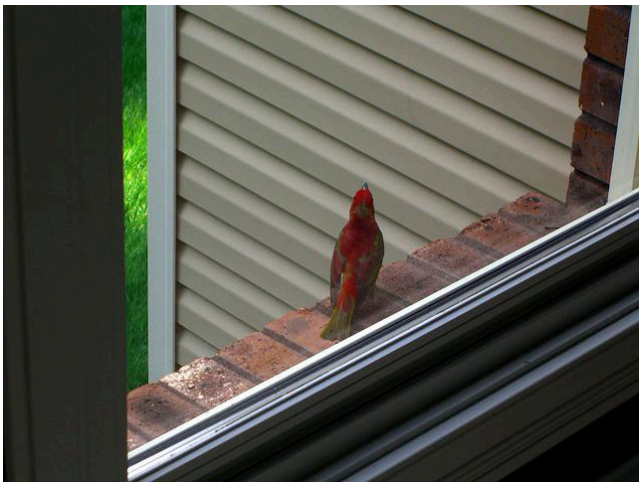
```
Calculating Predictions: 100%|          | 1/1 [00:00<00:00,  5.99it/s]
Inception Score for Stable Diffusion Generated Images with 2 Split: 1.7581 ± 0.2180
```

B. Image Generation Outputs of the Stable Diffusion Model

(The image on the left represents the generated output, while the image on the right displays the corresponding ground truth.)

C. Image Generation Outputs of the PixArt Model

(The image on the left represents the generated output, while the image on the right displays the corresponding ground truth.)



D. Score Outputs of the PixArt Model

```
CLIP Score: 0.3491, for caption: A flock of brown ducks swimming together on a lake.
CLIP Score: 0.3835, for caption: A red bird sitting on a window sill outside.
CLIP Score: 0.3333, for caption: A table covered in stars and a bowl filled with meat and vegetables.
CLIP Score: 0.2791, for caption: A white plate topped with bread covered in leafy greens.
CLIP Score: 0.2996, for caption: A desk with an open laptop computer next to a desktop computer
```

```
Calculating Inception Score...
Calculating Predictions: 100%|██████████| 1/1 [00:00<00:00,  2.14it/s]
Inception Score for Stable Diffusion Generated Images with 2 Split: 1.7641 ± 0.0937
```

```
Caption: A table covered in stars and a bowl filled with meat and vegetables.
FID Score (Text-to-Image): 126.5457
```

```
Caption: A white plate topped with bread covered in leafy greens.
FID Score (Text-to-Image): 83.1504
```

```
Caption: A desk with an open laptop computer next to a desktop computer.
FID Score (Text-to-Image): 95.7499
```

```
Caption: A red bird sitting on a window sill outside.
FID Score (Text-to-Image): 131.3837
```

```
Caption: A flock of brown ducks swimming together on a lake.
FID Score (Text-to-Image): 138.3201
```

E. Additional Python Code

The detailed codes used for the aforementioned models can be accessed from the drive link provided below.

https://drive.google.com/drive/folders/1QNOE19DZob5M2TAc1SOOz301vY_FZKah

```
import os

import json

import shutil

import random

from tqdm import tqdm



# --- Paths ---

coco_images_path = '/content/train2017'  # Path to COCO images

coco_annotations_path = '/content/annotations/captions_train2017.json'  # Captions annotation file



output_dir = '/content/output/'  # Directory where train/val/test folders will be saved

os.makedirs(output_dir, exist_ok=True)



# Paths for train, validation, and test splits

train_images_path = os.path.join(output_dir, 'train2017')

val_images_path = os.path.join(output_dir, 'val2017')

test_images_path = os.path.join(output_dir, 'test2017')



# Create directories for image splits

os.makedirs(train_images_path, exist_ok=True)

os.makedirs(val_images_path, exist_ok=True)

os.makedirs(test_images_path, exist_ok=True)



# --- Load COCO Captions JSON ---
```

```
with open(coco_annotations_path, 'r') as f:
    coco_data = json.load(f)

images = coco_data['images']  # List of all images
annotations = coco_data['annotations']  # List of all annotations

# --- Shuffle and Split Data ---
random.seed(42)  # For reproducibility
random.shuffle(images)

num_images = len(images)
train_split = int(0.8 * num_images)
val_split = int(0.9 * num_images)

train_images = images[:train_split]
val_images = images[train_split:val_split]
test_images = images[val_split:]

print(f"Total images: {num_images}, Train: {len(train_images)}, Val:
{len(val_images)}, Test: {len(test_images)}")

# --- Split Captions ---
def get_annotations_for_images(images_subset, annotations):
    image_ids = {img['id'] for img in images_subset}
    return [ann for ann in annotations if ann['image_id'] in image_ids]

train_annotations = get_annotations_for_images(train_images, annotations)
val_annotations = get_annotations_for_images(val_images, annotations)
test_annotations = get_annotations_for_images(test_images, annotations)
```

```
# --- Organize Captions into JSON-Friendly Format ---
def format_data(images_subset, annotations_subset):
    formatted_data = {
        "images": images_subset,
        "annotations": []
    }
    for ann in annotations_subset:
        formatted_data["annotations"].append({
            "image_id": ann['image_id'],
            "caption": ann['caption'],
            "id": ann['id']  # Ensure we retain the annotation ID for traceability
        })
    return formatted_data

# --- Save JSON Files ---
def save_json(data, path):
    with open(path, 'w') as f:
        json.dump(data, f, indent=4)

train_data = format_data(train_images, train_annotations)
val_data = format_data(val_images, val_annotations)
test_data = format_data(test_images, test_annotations)

save_json(train_data, os.path.join(output_dir, 'captions_train2017.json'))
save_json(val_data, os.path.join(output_dir, 'captions_val2017.json'))
save_json(test_data, os.path.join(output_dir, 'captions_test2017.json'))

# --- Copy Images to Corresponding Directories ---
```

```
def copy_images(images_subset, src_folder, dest_folder):
    for img_info in tqdm(images_subset, desc=f"Copying to {dest_folder}"):
        img_filename = img_info['file_name']
        src_path = os.path.join(src_folder, img_filename)
        dest_path = os.path.join(dest_folder, img_filename)
        shutil.copy(src_path, dest_path)

copy_images(train_images, coco_images_path, train_images_path)
copy_images(val_images, coco_images_path, val_images_path)
copy_images(test_images, coco_images_path, test_images_path)

print("Data split and files copied successfully!")
```

```
import os
import model_loader
import pipeline
from PIL import Image
from transformers import CLIPTokenizer
import torch
import json
from tqdm import tqdm

captions_path =
r"C:\Users\Eray\Desktop\imagen1\annotations\captions_train2017.json"
images_path = r"C:\Users\Eray\Desktop\RAT-GAN\train2017"
generated_images_dir =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images"
ground_truth_images_dir =
r"C:\Users\Eray\Desktop\stablediffusionfıd\ground_truth_images"
```

```
DEVICE = "cuda"

ALLOW_CUDA = True

ALLOW_MPS = False

if torch.cuda.is_available() and ALLOW_CUDA:

    DEVICE = "cuda"

elif (torch.has_mps or torch.backends.mps.is_available()) and
ALLOW_MPS:

    DEVICE = "mps"

print(f"Using device: {DEVICE}")



tokenizer = CLIPTokenizer("../data/vocab.json",
merges_file="../data/merges.txt")

model_file = "../data/v1-5-pruned-emaonly.ckpt"

models = model_loader.preload_models_from_standard_weights(model_file,
DEVICE)


with open(captions_path, 'r') as f:

    captions_data = json.load(f)



captions_dict = {}

for annotation in captions_data['annotations']:

    image_id = annotation['image_id']

    caption = annotation['caption']

    if image_id not in captions_dict:

        captions_dict[image_id] = []

    captions_dict[image_id].append(caption)
```

```
do_cfg = True
cfg_scale = 5
sampler = "ddpm"
num_inference_steps = 1
seed = 42



os.makedirs(generated_images_dir, exist_ok=True)
os.makedirs(ground_truth_images_dir, exist_ok=True)



import random

def generate_and_save_images(num_images):

    selected_items = list(captions_dict.items())
    random.shuffle(selected_items)



    selected_items = selected_items[:num_images]

    for i, (image_id, captions) in tqdm(enumerate(selected_items),
total=num_images, desc="Generating images"):
        prompt = captions[0]
        print(f"Generating image {i + 1}/{num_images} with caption:
{prompt}")
```

```
        ground_truth_image_path = os.path.join(images_path,
f"{str(image_id).zfill(12)}.jpg")
        if os.path.exists(ground_truth_image_path):
            gt_image =
Image.open(ground_truth_image_path).convert("RGB")
            gt_image.save(os.path.join(ground_truth_images_dir,
f"ground_truth_{image_id}.jpg"))



        output_image = pipeline.generate(
            prompt=prompt,
            uncond_prompt="",
            input_image=None,
            strength=0.9,
            do_cfg=do_cfg,
            cfg_scale=cfg_scale,
            sampler_name=sampler,
            n_inference_steps=num_inference_steps,
            seed=seed + i,
            models=models,
            device=DEVICE,
            idle_device="cuda",
            tokenizer=tokenizer,
        )
        generated_image_path = os.path.join(generated_images_dir,
f"generated_{image_id}.png")
        image = Image.fromarray(output_image)
        image.save(generated_image_path)
```

```
num_images_to_generate = 5
generate_and_save_images(num_images_to_generate)
```

```
import torch
import clip
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)

def compute_clip_score(image_path, text_prompt):

    image = preprocess(Image.open(image_path)).unsqueeze(0).to(device)

    text = clip.tokenize([text_prompt]).to(device)
    with torch.no_grad():
        image_features = model.encode_image(image)
        text_features = model.encode_text(text)
```

```
    image_features /= image_features.norm(dim=-1, keepdim=True)
    text_features /= text_features.norm(dim=-1, keepdim=True)

    similarity = (image_features @ text_features.T).item()

    return similarity

image_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images\duck.png"
text_prompt = "A flock of brown ducks swimming together on a lake."
clip_score = compute_clip_score(image_path, text_prompt)
print(f"CLIP Score: {clip_score:.4f}, for caption: {text_prompt}")
image_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images\bird.png"
text_prompt = "A red bird sitting on a window sill outside."
clip_score = compute_clip_score(image_path, text_prompt)
print(f"CLIP Score: {clip_score:.4f}, for caption: {text_prompt}")
image_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images\table.png"
text_prompt = "A table covered in stars and a bowl filled with meat and
vegetables."
clip_score = compute_clip_score(image_path, text_prompt)
print(f"CLIP Score: {clip_score:.4f}, for caption: {text_prompt}")
image_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images\bread.png"
text_prompt = "A white plate topped with bread covered in leafy
greens."
clip_score = compute_clip_score(image_path, text_prompt)
```

```
print(f"CLIP Score: {clip_score:.4f}, for caption: {text_prompt}")

image_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images\laptop.png"

text_prompt = "A desk with an open laptop computer next to a desktop
computer"

clip_score = compute_clip_score(image_path, text_prompt)

print(f"CLIP Score: {clip_score:.4f}, for caption: {text_prompt}")


import torch

import torch.nn.functional as F

from torchvision.models import inception_v3

from torchvision import transforms

from PIL import Image

import os

import numpy as np

from tqdm import tqdm

generated_images_dir =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images"


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


inception = inception_v3(pretrained=True,
transform_input=False).to(device)

inception.eval()


preprocess = transforms.Compose([

    transforms.Resize((299, 299)),

    transforms.ToTensor(),

    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
```

```
])


def load_images(directory):
    images = []
    for img_file in os.listdir(directory):
        img_path = os.path.join(directory, img_file)
        if img_path.endswith(".png") or img_path.endswith(".jpg"):
            image = Image.open(img_path).convert("RGB")
            images.append(preprocess(image))
    return torch.stack(images)


def calculate_inception_score(images, splits=1):
    with torch.no_grad():
        preds = []
        for i in tqdm(range(0, len(images), 32), desc="Calculating
Predictions"):
            batch = images[i:i+32].to(device)
            pred = F.softmax(inception(batch), dim=1)
            preds.append(pred.cpu().numpy())
        preds = np.concatenate(preds, axis=0)


    split_scores = []
    for k in range(splits):
        part = preds[k * (len(preds) // splits):(k + 1) * (len(preds)
// splits), :]
        py = np.mean(part, axis=0)
        scores = [np.sum(p * np.log(p / py)) for p in part]
        split_scores.append(np.exp(np.mean(scores)))
```

```
    return np.mean(split_scores), np.std(split_scores)


print("Loading generated images...")

images = load_images(generated_images_dir)


for i, img in enumerate(images):

    print(f"Image {i + 1}: Shape: {img.shape}, Min: {img.min().item()},
Max: {img.max().item()}")


print("Calculating Inception Score...")

mean_is, std_is = calculate_inception_score(images, splits=2)

print(f"Inception Score for Stable Diffusion Generated Images with 2
Split: {mean_is:.4f} ± {std_is:.4f}")


import torch

from pytorch_fid import fid_score


def compute_text_to_image_fid(real_images_dir, generated_images_dir,
caption, batch_size=16):

    """

    real_images_dir: Path to directory containing real images
corresponding to text prompts.

    generated_images_dir: Path to directory containing images generated
from text prompts.

    caption: The text description used for image generation.

    """

    device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```
    fid_value = fid_score.calculate_fid_given_paths([real_images_dir,
generated_images_dir],


batch_size=batch_size,

                                                device=device,

                                                dims=2048)


    print(f"Caption: {caption}")

    print(f"FID Score (Text-to-Image): {fid_value:.4f}")

    return fid_value




caption = "A desk with an open laptop computer next to a desktop
computer."

real_images_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\ground_truth_images"  # Real
images (e.g., from MS COCO)

generated_images_path =
r"C:\Users\Eray\Desktop\stablediffusionfıd\generated_images"  #
Model-generated images

fid_score_value = compute_text_to_image_fid(real_images_path,
generated_images_path, caption, batch_size=64)


from PIL import Image

from IPython.display import display

import torch as th


from glide_text2im.download import load_checkpoint

from glide_text2im.model_creation import (

   create_model_and_diffusion,

   model_and_diffusion_defaults,
```

```
    model_and_diffusion_defaults_upsampler

)



# This notebook supports both CPU and GPU.

# On CPU, generating one sample may take on the order of 20 minutes.

# On a GPU, it should be under a minute.



has_cuda = th.cuda.is_available()

device = th.device('cpu' if not has_cuda else 'cuda')









from PIL import Image

import os

# Function to save and show the image

def save_and_show_images(up_samples, prompt, output_dir="generated_images"):

    # Ensure the output directory exists

    os.makedirs(output_dir, exist_ok=True)

    # Iterate through samples and save them

    for idx, sample in enumerate(up_samples):

        # Convert tensor to numpy and format it as an image

        image_array = ((sample.permute(1, 2, 0).cpu().numpy() + 1) *
127.5).astype('uint8')

        img = Image.fromarray(image_array)

        # Save the image using the prompt name or a unique identifier

        image_filename = f"{prompt.replace(' ', '_')}_{idx}.png"

        image_path = os.path.join(output_dir, image_filename)

        img.save(image_path)

        print(f"Image saved at {image_path}")

        # Optionally display the image

        img.show()

    return  image_path

# Directory to save images (change the path as desired)
```

```
output_directory = "custom_directory_for_images2"
import torch
import clip
from PIL import Image
generated_paths = []
average_total_clip = 0
for i in captions_text:
   # Sampling parameters
   prompt = i
   batch_size = 1
   guidance_scale = 3.0
   # Directory to save images (change the path as desired)
   output_directory = "custom_directory_for_images2"
   # Tune this parameter to control the sharpness of 256x256 images.
   # A value of 1.0 is sharper, but sometimes results in grainy artifacts.
   upsample_temp = 0.997
   ##############################
   # Sample from the base model #
   ##############################
   # Create the text tokens to feed to the model.
   tokens = model.tokenizer.encode(prompt)
   tokens, mask = model.tokenizer.padded_tokens_and_mask(
       tokens, option['text_ctx']
   )
   # Create the classifier-free guidance tokens (empty)
   full_batch_size = batch_size * 2
   uncond_tokens, uncond_mask = model.tokenizer.padded_tokens_and_mask(
       [], option['text_ctx']
   )
```

```
   # Pack the tokens together into model kwargs.
   model_kwargs = dict(
       tokens=th.tensor(
           [tokens] * batch_size + [uncond_tokens] * batch_size, device=device
       ),
```

```
        mask=th.tensor(
            [mask] * batch_size + [uncond_mask] * batch_size,
            dtype=th.bool,
            device=device,
        ),
    )
    # Create a classifier-free guidance sampling function
    def model_fn(x_t, ts, **kwargs):
        half = x_t[: len(x_t) // 2]
        combined = th.cat([half, half], dim=0)
        model_out = model(combined, ts, **kwargs)
        eps, rest = model_out[:, :3], model_out[:, 3:]
        cond_eps, uncond_eps = th.split(eps, len(eps) // 2, dim=0)
        half_eps = uncond_eps + guidance_scale * (cond_eps - uncond_eps)
        eps = th.cat([half_eps, half_eps], dim=0)
        return th.cat([eps, rest], dim=1)
    # Sample from the base model.
    model.del_cache()
    samples = diffusion.p_sample_loop(
        model_fn,
        (full_batch_size, 3, options["image_size"], options["image_size"]),
        device=device,
        clip_denoised=True,
        progress=True,
        model_kwargs=model_kwargs,
        cond_fn=None,
    )[:batch_size]
    model.del_cache()
    ##############################
    # Upsample the 64x64 samples #
    ##############################
    tokens = model_up.tokenizer.encode(prompt)
    tokens, mask = model_up.tokenizer.padded_tokens_and_mask(
        tokens, options_up['text_ctx']
    )
```

```
# Create the model conditioning dict.
model_kwargs = dict(
    # Low-res image to upsample.
    low_res=((samples+1)*127.5).round()/127.5 - 1,
    # Text tokens
    tokens=th.tensor(
        [tokens] * batch_size, device=device
    ),
    mask=th.tensor(
        [mask] * batch_size,
        dtype=th.bool,
        device=device,
    ),
)
# Sample from the base model.
model_up.del_cache()
up_shape = (batch_size, 3, options_up["image_size"], options_up["image_size"])
up_samples = diffusion_up.ddim_sample_loop(
    model_up,
    up_shape,
    noise=th.randn(up_shape, device=device) * upsample_temp,
    device=device,
    clip_denoised=True,
    progress=True,
    model_kwargs=model_kwargs,
    cond_fn=None,
)[:batch_size]
model_up.del_cache()
print(i)
# Show the output
show_images(up_samples)
image_path = save_and_show_images(up_samples, i, output_dir=output_directory)
# Example Usage
# Generating CLIP Score
clip_score = compute_clip_score(image_path, prompt)
```

```
    generated_paths.append(image_path)
    # Checking for Intensity socre
    average_total_clip += clip_score
    print(f"CLIP Score: {clip_score:.4f}")
# Average CLIP Score
average_total_clip /= len(caption_text)
print(f"Average CLIP Score: {average_total_clip:.4f}")
# Generating Diveristy Score
compute_diversity_score(generated_paths)
```

```
def convert_coco_to_data_info(coco_json_path, image_root,
output_json_path):
    with open(coco_json_path, 'r') as f:
        coco_data = json.load(f)

    image_metadata = {img['id']: img for img in coco_data['images']}

    # Prepare the `data_info.json` format
    data_info = []
    for ann in tqdm(coco_data['annotations']):
        image_id = ann['image_id']
        image_info = image_metadata[image_id]
        file_name = image_info['file_name']
        height = image_info['height']
        width = image_info['width']
        ratio = width / height
        prompt = ann['caption']

        data_info.append({
            "height": height,
```

```
            "width": width,
            "ratio": ratio,
            "path": file_name,
            "prompt": prompt,
            "sharegpt4v": prompt
        })

    with open(output_json_path, 'w') as f:
        json.dump(data_info, f, indent=4)

    print(f"Converted JSON saved to {output_json_path}")

convert_coco_to_data_info(
    coco_json_path="./coco_data/annotations/captions_train2017.json",
    image_root="./coco_data/train2017",
    output_json_path="./data_info.json"
)

def create_pixart_toy_dataset(coco_json_path, image_root,
base_path="./PixArt-sigma"):
    toy_dataset_path = os.path.join(base_path,
"pixart-sigma-toy-dataset")

    os.makedirs(toy_dataset_path, exist_ok=True)
    intern_imgs_path = os.path.join(toy_dataset_path, "InternImgs")
     intern_test_imgs_path = os.path.join(toy_dataset_path,
"InternTestImgs")
    intern_data_path = os.path.join(toy_dataset_path, "InternData")
    os.makedirs(intern_imgs_path, exist_ok=True)
```

```
    os.makedirs(intern_test_imgs_path, exist_ok=True)
    os.makedirs(intern_data_path, exist_ok=True)



    with open(coco_json_path, 'r') as f:
        coco_data = json.load(f)


    val_test_ratio = 0.1
    val_test_data_size = int(len(coco_data) * val_test_ratio)
    val_data_size = val_test_data_size // 2


    train_data, val_test_data = train_test_split(coco_data,
test_size=val_test_data_size, random_state=42)
    val_data, test_data = train_test_split(val_test_data,
test_size=0.5, random_state=42)


    val_data_info_path = os.path.join(intern_data_path,
"data_info.json")
    test_data_info_path = os.path.join(intern_data_path,
"test_data_info.json")


    with open(val_data_info_path, 'w') as f:
        json.dump(val_data, f, indent=4)
    print(f"Validation data saved to {val_data_info_path}
({len(val_data)} samples).")


    with open(test_data_info_path, 'w') as f:
        json.dump(test_data, f, indent=4)
    print(f"Test data saved to {test_data_info_path} ({len(test_data)}
samples).")
```

```
    val_image_ids = {entry["path"] for entry in val_data}
    print(f"Moving {len(val_image_ids)} validation images to
{intern_imgs_path}...")
    for img_name in val_image_ids:
        src_img_path = os.path.join(image_root, img_name)
        dest_img_path = os.path.join(intern_imgs_path, img_name)
        if os.path.exists(src_img_path):
            shutil.copy(src_img_path, dest_img_path)

    test_image_ids = {entry["path"] for entry in test_data}
    print(f"Moving {len(test_image_ids)} test images to
{intern_test_imgs_path}...")
    for img_name in test_image_ids:
        src_img_path = os.path.join(image_root, img_name)
        dest_img_path = os.path.join(intern_test_imgs_path, img_name)
        if os.path.exists(src_img_path):
            shutil.copy(src_img_path, dest_img_path)


    print(f'Moved validation images to "{intern_imgs_path}" and test
images to "{intern_test_imgs_path}".')
    print(f'Created "InternImgs", "InternTestImgs", and "InternData"
with validation/test splits.')

create_pixart_toy_dataset(
    coco_json_path="./data_info.json",  # Path to the converted
data_info.json
    image_root="./coco_data/train2017",  # Path to COCO 2017 images
    base_path="./PixArt-sigma"  # Base directory for toy dataset
```

```
)
```

```
import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader

import torchvision.transforms as transforms

from torchvision.datasets import CocoCaptions

import clip

from PIL import Image

import numpy as np

from torch.optim import Adam

import json

import os

from tqdm import tqdm


# Dataset Class

class CocoDataset(Dataset):

   def __init__(self, root, annFile, split='train', transform=None):

       self.coco = CocoCaptions(root=root, annFile=annFile)

       self.transform = transform


       # Create train/val/test split indices

       total_size = len(self.coco)

       indices = list(range(total_size))

       np.random.shuffle(indices)


       train_size = int(0.8 * total_size)
```

```
        val_size = int(0.1 * total_size)
        test_size = int(0.1 * total_size)

        if split == 'train':
            self.indices = indices[:train_size]
        elif split == 'val':
            self.indices = indices[train_size:train_size + val_size]
        elif  split == 'test':
            self.indices = indices[train_size + val_size:]

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        true_idx = self.indices[idx]
        image, captions = self.coco[true_idx]

        if self.transform:
            image = self.transform(image)

        caption = np.random.choice(captions)
        return image, caption

# Generator Architecture
class Generator(nn.Module):
    def __init__(self, latent_dim, text_embedding_dim):
        super(Generator, self).__init__()
        self.latent_dim = latent_dim
        self.text_embedding_dim = text_embedding_dim
```

```
        # Text embedding projection
        self.text_projection = nn.Linear(text_embedding_dim, text_embedding_dim)

        # Initial dense layer
        self.initial = nn.Linear(latent_dim + text_embedding_dim, 7 * 7 * 512)

        # Transposed convolution layers for 224x224 output
        self.conv_blocks = nn.ModuleList([
            nn.Sequential(
                nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True)
            ),
            nn.Sequential(
                nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True)
            ),
            nn.Sequential(
                nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True)
            ),
            nn.Sequential(
                nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True)
            ),
```

```
            nn.Sequential(
                nn.ConvTranspose2d(32, 3, kernel_size=4, stride=2, padding=1),
                nn.Tanh()
            )
        ])

    def forward(self, z, text_embedding):
        # Project text embedding
        text_embedding = self.text_projection(text_embedding)

        # Concatenate noise and text embedding
        x = torch.cat([z, text_embedding], dim=1)

        # Initial dense layer
        x = self.initial(x)
        x = x.view(-1, 512, 7, 7)

        # Apply transposed convolution blocks
        for conv_block in self.conv_blocks:
            x = conv_block(x)

        return x

# Discriminator Architecture
class Discriminator(nn.Module):
    def __init__(self, text_embedding_dim):
        super(Discriminator, self).__init__()
        self.text_embedding_dim = text_embedding_dim
```

```
        # Image processing layers
        self.conv_blocks = nn.ModuleList([
            nn.Sequential(
                nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(0.25)
            ),
            nn.Sequential(
                nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(128),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(0.25)
            ),
            nn.Sequential(
                nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(256),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(0.25)
            ),
            nn.Sequential(
                nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(512),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(0.25)
            ),
            nn.Sequential(
                nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(512),
                nn.LeakyReLU(0.2, inplace=True),
```

```
                nn.Dropout2d(0.25)
            )
        ])

        # Text embedding projection
        self.text_projection = nn.Linear(text_embedding_dim, 512)

        # Calculate feature size for flattened output
        self.feature_size = 512 * 7 * 7  # For 224x224 input

        # Feature combination layer
        self.feature_combine = nn.Sequential(
            nn.Linear(self.feature_size + 512, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        # Final classification layer
        self.final = nn.Sequential(
            nn.Linear(1024, 1),
            nn.Sigmoid()
        )

    def get_features(self, image, text_embedding):
        # Process image through conv blocks
        x = image
        for conv_block in self.conv_blocks:
            x = conv_block(x)
        return x.view(-1, self.feature_size)
```

```
    def forward(self, image, text_embedding):
        # Get image features
        image_features = self.get_features(image, text_embedding)

        # Project text embedding
        text_features = self.text_projection(text_embedding)

        # Combine features
        combined = torch.cat([image_features, text_features], dim=1)
        combined = self.feature_combine(combined)

        # Final classification
        return self.final(combined)

# CLIP-GAN Class
class CLIPGAN:
    def __init__(self, latent_dim, device):
        self.latent_dim = latent_dim
        self.device = device

        # Load CLIP model
        self.clip_model, self.clip_preprocess = clip.load("ViT-B/32", device=device)

        # Initialize generator and discriminator
        self.generator = Generator(latent_dim, 512).to(device)  # CLIP embedding dim
is 512
        self.discriminator = Discriminator(512).to(device)
```

```
        # Setup optimizers with different learning rates (TTUR)
        self.g_optimizer = Adam(self.generator.parameters(), lr=0.0001, betas=(0.5,
0.999))
        self.d_optimizer = Adam(self.discriminator.parameters(), lr=0.00001,
betas=(0.5, 0.999))

        # Loss functions
        self.adversarial_loss = nn.BCELoss()
        self.clip_loss = nn.CosineSimilarity()

        # Instance noise - decreases over time
        self.initial_noise_std = 0.1
        self.noise_decay = 0.995

    def train_step(self, real_images, captions, current_batch):
        batch_size = real_images.size(0)

        # Calculate current noise standard deviation
        noise_std = self.initial_noise_std * (self.noise_decay ** current_batch)

        # Smoother label range
        real_labels = torch.rand(batch_size, 1).to(self.device) * 0.2 + 0.8
        fake_labels = torch.rand(batch_size, 1).to(self.device) * 0.2

        # Convert real images to float32
        real_images = real_images.to(dtype=torch.float32)

        # Add instance noise to real images
        real_images = real_images + noise_std * torch.randn_like(real_images)
```

```
        # Encode text using CLIP and convert to float32
        with torch.no_grad():
            text_features =
self.clip_model.encode_text(clip.tokenize(captions).to(self.device))
            text_features = text_features.to(dtype=torch.float32)

        # Train Generator every other batch
        if current_batch % 2 == 0:
            self.g_optimizer.zero_grad()

            # Generate fake images
            z = torch.randn(batch_size, self.latent_dim).to(self.device)
            fake_images = self.generator(z, text_features)
            fake_images = fake_images + noise_std * torch.randn_like(fake_images)

            # Generator loss
            fake_pred_g = self.discriminator(fake_images, text_features)
            g_adv_loss = self.adversarial_loss(fake_pred_g, real_labels)

            # CLIP similarity loss
            with torch.no_grad():
                fake_image_features = self.clip_model.encode_image(fake_images)
                fake_image_features = fake_image_features.to(dtype=torch.float32)
            clip_similarity = self.clip_loss(fake_image_features,
text_features).mean()

            # Total generator loss
            g_loss = g_adv_loss - 0.05 * clip_similarity
            g_loss.backward()
```

```
            # Gradient clipping
            torch.nn.utils.clip_grad_norm_(self.generator.parameters(),
max_norm=1.0)
            self.g_optimizer.step()
        else:
            g_loss = torch.tensor(0.0).to(self.device)
            clip_similarity = torch.tensor(0.0).to(self.device)
            fake_images = None

        # Train Discriminator when accuracy < 0.85
        if fake_images is None:
            z = torch.randn(batch_size, self.latent_dim).to(self.device)
            fake_images = self.generator(z, text_features)
            fake_images = fake_images + noise_std * torch.randn_like(fake_images)

        with torch.no_grad():
            real_pred = self.discriminator(real_images, text_features)
            fake_pred = self.discriminator(fake_images.detach(), text_features)
            d_acc = 0.5 * (torch.mean((real_pred > 0.5).float()) +
                           torch.mean((fake_pred < 0.5).float()))

        # Calculate D loss
        self.d_optimizer.zero_grad()

        real_pred = self.discriminator(real_images, text_features)
        d_real_loss = self.adversarial_loss(real_pred, real_labels)

        fake_pred = self.discriminator(fake_images.detach(), text_features)
        d_fake_loss = self.adversarial_loss(fake_pred, fake_labels)
```

```
        # Feature matching loss
        with torch.no_grad():
            real_features = self.discriminator.get_features(real_images,
text_features)
        fake_features = self.discriminator.get_features(fake_images.detach(),
text_features)
        feature_matching_loss = F.mse_loss(fake_features, real_features)


        d_loss = d_real_loss + d_fake_loss + 0.1 * feature_matching_loss


        if d_acc < 0.85:
            d_loss.backward()
            torch.nn.utils.clip_grad_norm_(self.discriminator.parameters(),
max_norm=1.0)
            self.d_optimizer.step()


        return {
            'd_loss': d_loss.item(),
            'g_loss': g_loss.item(),
            'clip_similarity': clip_similarity.item(),
            'd_acc': d_acc.item(),
            'noise_std': noise_std
        }


import matplotlib.pyplot as plt


def validate(clip_gan, val_loader, device):
    clip_gan.generator.eval()
    clip_gan.discriminator.eval()
```

```
    total_d_loss = 0
    total_g_loss = 0
    total_clip_sim = 0
    num_batches = 0

    with torch.no_grad():
        for batch_idx, (images, captions) in enumerate(val_loader):
            images = images.to(device)
            batch_size = images.size(0)

            # Encode text using CLIP
            text_features =
clip_gan.clip_model.encode_text(clip.tokenize(captions).to(device))
            text_features = text_features.to(dtype=torch.float32)

            # Generate fake images
            z = torch.randn(batch_size, clip_gan.latent_dim).to(device)
            fake_images = clip_gan.generator(z, text_features)

            # Discriminator predictions
            real_pred = clip_gan.discriminator(images, text_features)
            fake_pred = clip_gan.discriminator(fake_images, text_features)

            # Calculate losses
            real_labels = torch.ones(batch_size, 1).to(device) * 0.9
            fake_labels = torch.zeros(batch_size, 1).to(device) + 0.1

            d_real_loss = clip_gan.adversarial_loss(real_pred, real_labels)
```

```
            d_fake_loss = clip_gan.adversarial_loss(fake_pred, fake_labels)
            d_loss = (d_real_loss + d_fake_loss) / 2

            # Generator and CLIP loss
            g_loss = clip_gan.adversarial_loss(fake_pred, real_labels)
            fake_image_features = clip_gan.clip_model.encode_image(fake_images)
            fake_image_features = fake_image_features.to(dtype=torch.float32)
            clip_similarity = clip_gan.clip_loss(fake_image_features,
text_features).mean()

            total_d_loss += d_loss.item()
            total_g_loss += g_loss.item()
            total_clip_sim += clip_similarity.item()
            num_batches += 1

    clip_gan.generator.train()
    clip_gan.discriminator.train()

    return {
        'd_loss': total_d_loss / num_batches,
        'g_loss': total_g_loss / num_batches,
        'clip_sim': total_clip_sim / num_batches
    }

def plot_losses(train_history, val_history, save_dir):
    plt.figure(figsize=(15, 5))
    epochs = range(len(train_history['d_loss']))

    # Plot discriminator loss
```

```
    plt.subplot(131)
    plt.plot(epochs, train_history['d_loss'], label='Train D Loss')
    plt.plot(epochs, val_history['d_loss'], label='Val D Loss')
    plt.title('Discriminator Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    # Plot generator loss
    plt.subplot(132)
    plt.plot(epochs, train_history['g_loss'], label='Train G Loss')
    plt.plot(epochs, val_history['g_loss'], label='Val G Loss')
    plt.title('Generator Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    # Plot CLIP similarity
    plt.subplot(133)
    plt.plot(epochs, train_history['clip_sim'], label='Train CLIP Sim')
    plt.plot(epochs, val_history['clip_sim'], label='Val CLIP Sim')
    plt.title('CLIP Similarity')
    plt.xlabel('Epoch')
    plt.ylabel('Similarity')
    plt.legend()
    plt.grid(True)
```

```
    plt.tight_layout()
    plt.savefig(os.path.join(save_dir, 'training_history.png'))
    plt.close()


def train(clip_gan, train_loader, val_loader, num_epochs, device, save_dir):
    os.makedirs(save_dir, exist_ok=True)


    train_history = {
        'd_loss': [],
        'g_loss': [],
        'clip_sim': []
    }
    val_history = {
        'd_loss': [],
        'g_loss': [],
        'clip_sim': []
    }


    for epoch in range(num_epochs):
        # Training
        total_d_loss = 0
        total_g_loss = 0
        total_clip_sim = 0
        num_g_updates = 0


        for batch_idx, (images, captions) in enumerate(tqdm(train_loader)):
            images = images.to(device)
```

```
            losses = clip_gan.train_step(images, captions, batch_idx + epoch *
len(train_loader))


            total_d_loss += losses['d_loss']
            if batch_idx % 2 == 0:
                total_g_loss += losses['g_loss']
                total_clip_sim += losses['clip_similarity']
                num_g_updates += 1


            if batch_idx % 100 == 0:
                print(f"Epoch [{epoch}/{num_epochs}] "
                      f"Batch [{batch_idx}/{len(train_loader)}] "
                      f"D_loss: {losses['d_loss']:.4f} "
                      f"G_loss: {losses['g_loss']:.4f} "
                      f"CLIP_sim: {losses['clip_similarity']:.4f} "
                      f"D_acc: {losses['d_acc']:.4f} "
                      f"Noise: {losses['noise_std']:.4f}")


        # Calculate training epoch averages
        avg_train_d_loss = total_d_loss / len(train_loader)
        avg_train_g_loss = total_g_loss / num_g_updates
        avg_train_clip_sim = total_clip_sim / num_g_updates


        # Validation
        val_losses = validate(clip_gan, val_loader, device)


        # Update history
        train_history['d_loss'].append(avg_train_d_loss)
        train_history['g_loss'].append(avg_train_g_loss)
```

```
        train_history['clip_sim'].append(avg_train_clip_sim)


        val_history['d_loss'].append(val_losses['d_loss'])
        val_history['g_loss'].append(val_losses['g_loss'])
        val_history['clip_sim'].append(val_losses['clip_sim'])


        # Save models
        if (epoch + 1) % 5 == 0:
            torch.save({
                'generator_state_dict': clip_gan.generator.state_dict(),
                'discriminator_state_dict': clip_gan.discriminator.state_dict(),
                'train_history': train_history,
                'val_history': val_history
            }, os.path.join(save_dir, f'checkpoint_epoch_{epoch+1}.pt'))


        print(f"\nEpoch [{epoch}/{num_epochs}] "
              f"Train D_loss: {avg_train_d_loss:.4f} "
              f"Train G_loss: {avg_train_g_loss:.4f} "
              f"Train CLIP_sim: {avg_train_clip_sim:.4f} "
              f"Val D_loss: {val_losses['d_loss']:.4f} "
              f"Val G_loss: {val_losses['g_loss']:.4f} "
              f"Val CLIP_sim: {val_losses['clip_sim']:.4f}\n")


    # Plot losses after training is complete
    plot_losses(train_history, val_history, save_dir)
    return train_history, val_history


def main():
    # Hyperparameters
```

```
    latent_dim = 100
    batch_size = 32
    num_epochs = 50
    image_size = 224  # Required for CLIP
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Dataset paths
    root = '/content/train2017'
    annFile = '/content/annotations/captions_train2017.json'

    # Transform for images - using CLIP's preprocessing
    transform = transforms.Compose([
        transforms.Resize(224, interpolation=transforms.InterpolationMode.BICUBIC),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize((0.48145466, 0.4578275, 0.40821073),
                             (0.26862954, 0.26130258, 0.27577711))
    ])

    # Create datasets
    train_dataset = CocoDataset(root, annFile, split='train', transform=transform)
    val_dataset = CocoDataset(root, annFile, split='val', transform=transform)
    test_dataset = CocoDataset(root, annFile, split='test', transform=transform)

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
num_workers=4)

    # Initialize CLIP-GAN
```

```
    clip_gan = CLIPGAN(latent_dim, device)



    # Train model

    save_dir = 'checkpoints'

    train(clip_gan, train_loader, val_loader, num_epochs, device, save_dir)



if __name__ == '__main__':

    main()
```