**Project Stage-II**

**on**

**Generating Image Variations from Text**

**Submitted**

in partial fulfilment of the requirements for

the award of the degree of

**Bachelor of Technology**

in

**Computer Science and Engineering**

**(Artificial Intelligence and Machine Learning)**

By

**AYMAN MOHAMMED (20261A6607)**

**TIPPARAJU VENKATA SAI ANIRUDH (20261A6653)**

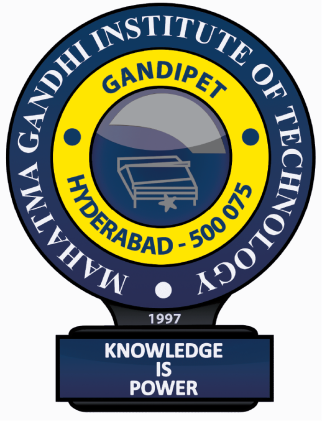
Under the Guidance of

**Dr. K. Mahesh Kumar**

Associate Professor

**Ms. B. Varalaxmi**

Asst. Professor



**DEPT. OF EMERGING TECHNOLOGIES**

**MAHATMA GANDHI INSTITUTE OF TECHNOLOGY(Autonomous)**

(Affiliated to Jawaharlal Nehru Technological University, Hyderabad)

Gandipet, Hyderabad-500075, Telangana, INDIA

**2023 – 2024**

**MAHATMA GANDHI INSTITUTE OF TECHNOLOGY(A)**

**(Affiliated to Jawaharlal Nehru Technological University Hyderabad)**

**GANDIPET, HYDERABAD, TELANGANA– 500 075 (INDIA)**

**CERTIFICATE**



This is to certify that the project entitled **Generating Image Variations from Text** is being submitted by **Mr. AYMAN MOHAMMED** bearing **Roll No: 20261A6607** and **Mr. TIPPARAJU VENKATA SAI ANIRUDH** bearing **Roll No: 20261A6653** in partial fulfilment for the award of **Bachelor of Technology** in **Computer Science and Engineering (Artificial Intelligence and Machine Learning)** to **Jawaharlal Nehru Technological University, Hyderabad** is a record of bonafide work carried out by them under our guidance and supervision.

The results embodied in this project have not been submitted to any other University or Institute for the award of any degree or diploma.

**Project Guides**  **Head of the Department**

Dr. K. Mahesh Kumar

Associate Professor

Ms. B. Varalaxmi

Assistant Professor

Dr. M. Rama Bai

Professor

**EXTERNAL EXAMINER**

**DECLARATION**

This is to certify that the work reported in this project titled **Generating Image Variations from Text** is a record of work done by us in the Department of Emerging Technologies, Mahatma Gandhi Institute of Technology, Hyderabad.

No part of the work is copied from books/journals/internet and wherever the portion is taken, the same has been duly referred to in the text. The report is based on the work done entirely by us and not copied from any other source.

**AYMAN MOHAMMED TIPPARAJU VENKATA SAI ANIRUDH**

**(20261A6607) (20261A6653)**

**ACKNOWLEDGEMENT**

We would like to express our sincere thanks to **Dr. G. Chandra Mohan Reddy, Principal MGIT**, for providing the working facilities in college.

We wish to express our sincere thanks and gratitude to **Dr. Rama Bai, Professor and HoD**, Department of ET, MGIT, for all the timely support and valuable suggestions during the period of project.

We are also extremely thankful to our Project Coordinators, **Mr. R. Srinivas, Assistant Professor, Dept. of ET and Dr. D. Koteswara Rao, Assistant Professor, Dept. of ET,** for their valuable suggestions and interest throughout the course of this project

We are extremely thankful to our internal guides **Dr. K. Mahesh Kumar, Associate Professor**, Department of ET, MGIT and **Ms. B. Varalaxmi, Assistant Professor,** Department of ET, MGIT for their constant guidance, encouragement and moral support throughout the project.

Finally, we would also like to thank all the faculty and staff of the ET Department who helped us directly or indirectly, for completing this project.

**AYMAN MOHAMMED TIPPARAJU VENKATA SAI ANIRUDH**

**(20261A6607) (20261A6653)**

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| **TOPIC** | **PAGE NO.** |
| **Certificate**  **Declaration**  **Acknowledgement**  **List of figures**  **List of tables**  **Abstract**   1. **Introduction**    1. Problem Definition    2. Existing System    3. Proposed System    4. Scope    5. Requirements Specifications       1. Software Requirements       2. Hardware Requirements      1. **Literature Survey** 2. **Design and Methodology**    1. Architecture       1. Variable Autoencoder       2. CLIP Encoder       3. UNet       4. DDPM    2. Data Collection and Augmentation    3. Training the model    4. Model evaluation    5. Web App 3. **Implementation**    1. Variational Autoencoder       1. Encoder          1. VAE\_Encoder       2. Decoder          1. VAE\_AttentionBlock          2. VAE\_ResidualBlock          3. VAE\_Decoder    2. CLIP Encoder       1. CLIPEmbedding       2. CLIPLayer       3. CLIP    3. UNet       1. Time Embedding       2. UNet\_ResidualBlock       3. Unet\_AttentionBlock       4. Upsample       5. SwitchSequential       6. UNet       7. UNet\_OutputLayer       8. Diffusion    4. DDPM 4. **Results** 5. **Conclusion**   **References**  **Appendix** | i  ii  iii  vi  vii  viii  1  1  3  3  4  5  5  5  6  10  10  11  13  15  16  17  18  20  21  22  23  23  23  24  24  24  24  24  24  25  25  26  26  26  26  26  26  27  27  27  28  29  32 |
|  |  |
|  |  |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| Figure No. | Figure Name | Page No. |
| Figure 3.1 | Latent Diffuser model for inference | 10 |
| Figure 3.2 | Autoencoder diagram and it’s loss function | 11 |
| Figure 3.3 | Difference between an autoencoder and a variational autoencoder | 12 |
| Figure 3.4 | Variational autoencoder used in probabilistic diffusion model | 13 |
| Figure 3.5 | Contrastive Pretraining for CLIP | 14 |
| Figure 3.6 | Numpy-like pseudocode for its core implementation | 14 |
| Figure 3.7 | UNet model that has downsampling and upsampling with ResNet blocks | 15 |
| Figure 3.8 | Generating noise for the next sample using DDPM | 16 |
| Figure 3.9 | Generate noise for the sample at time t | 17 |
| Figure 3.10 | Noise removal for a given latent image by sampling from a Gaussian distribution | 17 |
| Figure 3.11 | Distribution of image sizes in LAOIN-400m. | 18 |
| Figure 3.12 | Image Preprocessing | 18 |
| Figure 3.13 | Sample image after adding noise to it | 18 |
| Figure 3.14 | Training Config for the model | 19 |
| Figure 3.15 | Last few epochs of training | 19 |
| Figure 3.16 | A measure of FID score of the model, compared with traditional models. | 20 |
| Figure 3.17 | Gradient scale vs IS | 20 |
| Figure 3.18  Figure 3.19 | Gradient Scale vs Precision, Recall  Web App Architecture | 21  21 |
| Figure 4.1  Figure 5.1  Figure 5.2  Figure 5.3  Figure 5.4  Figure 5.5 | High level image-to-image latent diffuser architecture  Initial screen for the web app  UI after uploading an image  Generated Image  Highlighting the masked part of the image  Generated image after applying the mask | 22  29  29  30  30  31 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| Table No. | Table Name | Page No. |
| Table 2.1 | Comparison of Literature Survey | 7 |
| Table 2.2 | DDPM Overview | 28 |

**ABSTRACT**

This project aims to develop a cutting-edge image generation system that leverages latent diffusers to create variations from text, enhancing the creative possibilities of digital art and design. The system will be built with a React frontend for user interaction and a Flask backend to manage the server-side operations. The project will also incorporate image inpainting capabilities, offering a significant upgrade over existing image generators

The key advantage of this project over existing Image generators Is IIbility to create variations from text, offering a level of creativity and flexibility that is not commonly found in other tools. By allowing users to input text prompts, assign weights to the importance of text, provide positive and negative prompts and generate a range of images based on those prompts, this system opens up new possibilities for artistic expression and design. Additionally, the integration of image inpainting capabilities enhances the system’s utility, making it a valuable tool for artists and designers alike. There are additional features that allow customisability of the output image apart from text such as image output dimensions i.e, the resolution, sampling steps in the image generation process etc.

The project’s significance lies in its potential to reshape multiple industries. From assisting designers in visualizing textual concepts with unprecedented fidelity to inspiring new avenues of artistic expression, the system’s applications are far- reaching. It can also serve as a tool for content creators to rapidly generate multimedia variations that align with specific textual cues.

1. **INTRODUCTION**

This project designed to revolutionize the field of image generation and inpainting. This project harnesses the power of advanced machine learning techniques, specifically focusing on Latent Diffusers and Convolutional Neural Networks (CNNs), to provide users with a unique and powerful tool for transforming images.

The project is built on the premise that image inpainting, the process of filling in missing or corrupted parts of images with plausible content, is not only a fascinating area of AI research but also a practical solution for a wide range of applications. From enhancing advertisements and social media posts to repairing old photos and creating entirely new images from scratch, our tool offers a versatile and user-friendly interface for both artists and enthusiasts alike.

At the heart of our project lies the Latent Diffuser model, a state-of-the-art AI technology that enables prompt-based inpainting without the need for manual masking. This technology allows users to specify the parts of an image they wish to modify or enhance, and the model then fills in the gaps with visually and semantically plausible content. This process is made even more accessible through our project, which provides a user-friendly interface for generating and editing images, making it possible for anyone to create stunning visuals with just a few clicks.

Whether you’re looking to fix a minor flaw in a photo, generate a new image from a text prompt, or explore the endless possibilities of what can be created with AI, our project is here to empower you.

* 1. **Problem Definition**

The objective of this project is to develop an advanced image transformation application, named Img2Img, that leverages cutting-edge technologies to transform text prompts into images, selectively modify images through inpainting, and enhance user experience with a seamless and intuitive interface. This application aims to provide users with the capability to generate images from textual descriptions, apply specific modifications to images, and interact with the software in a user-friendly manner.

Key Features and Requirements:

1. Image Variations: The application should be capable of generating images from textual prompts and another image given as input, allowing users to describe the desired image in detail.
2. Selection of Seed: Users should have the option to specify a seed value for the image generation process. This feature enables users to control the randomness of the output, allowing for the exploration of different generated outputs or the reproduction of specific images.
3. Inpainting: The application should include an inpainting feature that allows users to selectively modify images by filling in missing or damaged parts with content generated from textual prompts. This feature should support the use of arbitrary masks to define the areas to be inpainted and provide options for controlling the inpainting process.
4. Prompts: Users should be able to use prompts to guide the image generation and inpainting processes.
5. User Interface (UI): The application should feature a user interface that facilitates seamless use of the software. This includes intuitive navigation, clear instructions for each step, and the ability to easily adjust settings for text-to-image generation, inpainting, and other features. The UI should be designed to be accessible and user-friendly, ensuring a smooth workflow from prompt input to image generation and modification.

Challenges and Considerations:

1. Balancing Creativity and Control: Striking a balance between allowing users to explore creative possibilities and providing them with the tools to achieve specific outcomes is a critical challenge. The application must offer a range of options that cater to both casual users and those seeking precise control over the output.
2. Technical Complexity: The underlying technology, including the implementation of text-to-image and inpainting algorithms, introduces complexity in terms of performance, accuracy, and user experience. Ensuring that the application is both powerful and easy to use is a key consideration.
3. User Experience: Providing a seamless and enjoyable user experience is paramount. This includes responsive design, intuitive navigation, and a smooth workflow from prompt input to image generation and modification.
   1. **Existing System**

Existing systems that modify images have the following problems:

1. Limited Creativity: Existing image modification systems often offer a finite set of options, limiting users’ creativity and flexibility. Users are confined to predefined filters, effects, or adjustments, restricting their ability to explore unique modifications.
2. Dependency on Predefined Templates: Many image editing tools rely on predefined templates or styles, which may not always align with the user’s specific requirements or aesthetic preferences. This dependency restricts users from achieving the desired modifications, especially if their vision deviates from the available templates.
3. Inability to Address Complex Edits: Current systems struggle with complex image modifications that require intricate changes or precise adjustments. Users often face challenges when attempting to manipulate specific elements within an image, such as altering backgrounds, adjusting lighting conditions, or enhancing intricate details.
4. Limited Adaptability to Varied Input: Existing image editing software may not effectively handle diverse input images, such as those with unconventional compositions, varying lighting conditions, or complex textures. Consequently, users may encounter difficulties achieving desired modifications, particularly when working with unconventional or challenging input images.
5. Time-Consuming Manual Editing Processes: Traditional image editing methods often involve manual editing processes that are time-consuming and labor-intensive. Users may need to perform repetitive tasks or tedious adjustments, leading to inefficiencies and delays in achieving desired modifications.
   1. **Proposed System**

A generative model capable of doing image-to-image generation can address the limitations of existing image modification systems in several ways:

1. Enhanced Creativity through Generative Models: Img2img models can generate diverse and novel image modifications beyond the scope of predefined options. By learning from vast datasets, these models can produce a wide range of creative edits, allowing users to explore unique styles, effects, and alterations.
2. Flexible and Adaptive Modifications: Unlike traditional systems dependent on templates, img2img models can adapt to users’ preferences and input images. Through training on diverse datasets, these models learn to generate modifications tailored to specific user requirements, enabling greater flexibility and customization.
3. Complex Edits with Learned Representations: Img2img models can learn complex representations of images, enabling them to perform intricate modifications with precision. By understanding the underlying structure and content of images, these models can manipulate elements such as backgrounds, lighting, and details more effectively, facilitating sophisticated edits.
4. Handling Diverse Input with Robustness: Img2img models are trained on diverse datasets, allowing them to handle varied input images with greater robustness. Whether the input image features unconventional compositions, varying lighting conditions, or complex textures, the model can adapt its modifications to suit the characteristics of the input, ensuring consistent performance across different types of images.
5. Efficiency through Automated Editing: Img2img models automate the image editing process, reducing the need for manual intervention and tedious adjustments. By generating modifications directly from input images, these models streamline the editing workflow, saving time and effort for users while achieving desired results more efficiently.

Overall, generative models offer a promising solution to the limitations of existing image modification systems by leveraging advanced deep learning techniques to enable flexible, creative, and efficient editing capabilities.

* 1. **Scope**

Image-to-image models have a broad scope in the realm of image editing, offering a transformative approach to generating diverse and customizable modifications. By leveraging advanced deep learning techniques, such as generative adversarial networks (GANs), latetent diffusers(LD) these models excel in producing creative edits that go beyond the constraints of predefined options. From enhancing photos with unique styles and effects to performing intricate adjustments like background manipulation and detail enhancement, img2img models demonstrate versatility in addressing a wide range of editing needs. Moreover, their ability to adapt to diverse input images ensures robustness and consistency across various compositions, lighting conditions, and textures. With their capacity to automate editing processes, these models streamline workflows, saving time and effort while empowering users to achieve their desired modifications efficiently. Overall, img2img models represent a powerful tool for unlocking new possibilities in image editing, offering unparalleled flexibility, creativity, and efficiency.

* 1. **Requirements Specification**
     1. Software Requirements
        1. Language : Python
        2. Operating system : Windows or Linux
     2. Tools:
        1. PyTorch
     3. Hardware Requirements
        1. RAM – 16GB minimum
        2. GPU – 12GB vRAM minimum

1. **LITERATURE SURVEY**

This section shows the research work carried out on various optimizations in field of image to image generative models that have been made, in the following paragraphs. Various researchers have taken into account a plethora of mechanisms for deploying various models that make it appropriate for development.

**“Denoising Diffusion Probabilistic Models” [1]** was published by Jonathan Ho, Ajay Jain, Pieter Abbeel in 2020. The Denoising Diffusion Probabilistic Models (DDPM) paper introduces a new method for generative modelling, focusing on image processing. Unlike traditional approaches, DDPM employs denoising diffusion processes, gradually adding noise to images and training the model to reverse this process. This iterative denoising results in high-quality image generation, improved sample quality, better generalization, and enhanced stability during training. DDPM also incorporates probabilistic modelling for uncertainty estimation and sampling. This model has implications for computer vision, image synthesis, and data generation.

“**U-Net: Convolutional Networks for Biomedical Image Segmentation”** [2] was published by Olaf Ronneberger, Philipp Fischer, Thomas Brox in 2015. This paper introduces a novel architecture specifically designed for biomedical image segmentation tasks. The U-Net model consists of a contracting path for capturing context and a symmetric expanding path for precise localization. By combining these paths, U-Net effectively addresses challenges such as class imbalance and limited annotated data common in biomedical imaging. Its architecture allows for accurate segmentation of structures like cells and organs, making it a valuable tool in medical image analysis. U-Net has since become a cornerstone in various biomedical applications, including image segmentation, image registration, and disease diagnosis.

**“High-Resolution Image Synthesis with Latent Diffusion Models”** [3] was published by Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer in 2021.This paper presents a ground-breaking approach to high-fidelity image synthesis using Latent Diffusion Models (LDMs). LDMs leverage diffusion processes in latent space to generate high-resolution images with exceptional visual quality and fidelity. Unlike traditional generative models, LDMs use a diffusion process to iteratively transform a noise vector into a realistic image, allowing for the synthesis of highly detailed and diverse visual content. This method enables the generation of images with fine-grained details, sharp textures, and realistic structures, surpassing the capabilities of existing generative models. Additionally, LDMs offer controllable and coherent image synthesis, making them valuable for various applications such as image editing, content creation, and artistic expression.

“**Learning Transferable Visual Models From Natural Language Supervision**”[4] was published by Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, Ilya Sutskever in 2021. This paper introduces a pioneering approach to learning visual representations using natural language supervision. By leveraging large-scale datasets of paired images and text, the model learns to associate visual and textual information, facilitating cross-modal understanding. The method employs self-supervised learning techniques to train a visual encoder that maps images to a shared embedding space with textual representations. This enables the model to perform various downstream tasks, such as image classification and object detection, with improved generalization and transferability. Moreover, the learned visual representations exhibit semantic richness, capturing intricate relationships between visual and textual concepts. This approach offers a promising avenue for advancing multimodal learning and bridging the semantic gap between vision and language domains.

Table 2.1 showcases the literature survey.

**Table 2.1:** Literature Survey metadata along with the algorithms used, merits and demerits.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.NO** | **TITLE** | **AUTHORS** | **ALGORITHM / METHODOLOGY** | **MERITS OR ADVANTAGES** | **DEMERITS OR FUTURE SCOPE** |
| 1 | **Denoising Diffusion Probabilistic Models**  **(2020)** | Jonathan Ho, Ajay Jain, Pieter Abbeel | This method involves iteratively applying diffusion processes to a noise-corrupted version of an image and training a model to reverse this process to reconstruct the original image. | High-Quality Image Synthesis,  Generalization and Robustness | Research could focus on extending DDPM to handle other types of data beyond images, such as audio or video, |
| 2 | **U-Net: Convolutional Networks for Biomedical Image Segmentation**  (U-Net) | Olaf Ronneberger, Philipp Fischer, Thomas Brox | The skip connections between corresponding layers in the contracting and expanding paths help retain spatial information, facilitating accurate segmentation. | Accurate Segmentation,  Robustness to Limited Data. | Integrating attention mechanisms into U-Net could enhance its ability to focus on relevant image regions. |
| 3 | **High-Resolution Image Synthesis with Latent Diffusion Models** (2009) | Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer | Leverage diffusion processes in latent space to generate high-quality images | Realistic Image Generation, Flexibility in style and content | Computational Complexity |
| 4 | Learning Transferable Visual Models From Natural Language Supervision | Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, Ilya Sutskever | Learns visual representations using natural language supervision. | Cross-Modal Understanding,  Transferable Visual Representations | Computational Complexity |

1. **Design and Methodology**

**3.1 Architecture**

The image-to-image model is not just a single component, but multiple of them that are conjoined together by taking different inputs from different sources and producing a single output which is the modified image. Figure 3.1. shows the overview of the architecture.

A diagram of a schedule

Description automatically generated

**Figure 3.1:** Latent Diffuser model for inference

The encoder module takes the image as the input and produces an output which is known as the latent. This is the reason why the model is called a latent diffuser and instead of working in the probability distribution of the image dataset, it works in the probability distribution of the latent images.

The CLIP encoder takes in the input which is the text and uses positional embeddings to represent the text in terms of a text embedding that can be understood by the model. This is important as it has to learn to associate images with the text.

The time embeddings are also fed to the U-Net model. Time embeddings are nothing but embeddings of the current noise level on a particular time stamp. The U-Net model is fed this time embedding so it knows how much to “denoise”.

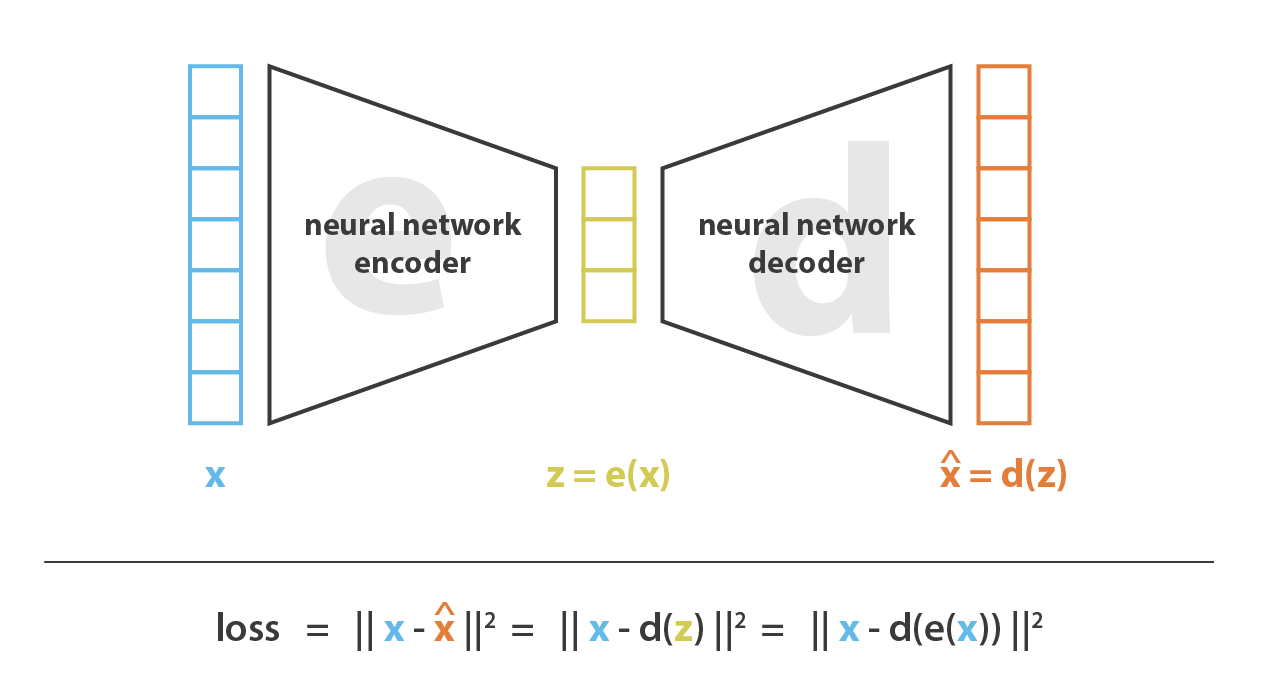
All these inputs are fed into the U-Net model as it denoises the image. The direction for denoising is described by the time embeddings and the text embeddings.

Once the U-Net predicts the probability distribution for denoising the image it is given to the Scheduler. The U-Net just predicts the noise of the image. Whereas it is the scheduler that actually denoises it and gives it back to the encoding to further denoise it.

Once all the series of denoising is done, the output is given to the decoder part of the variational autodecoder. This converts the image from the latent space to the actual image representation. The reason images are converted into their computational representations is to reduce computational complexity costs.

* 1. **.1 Variational Autoencoder**

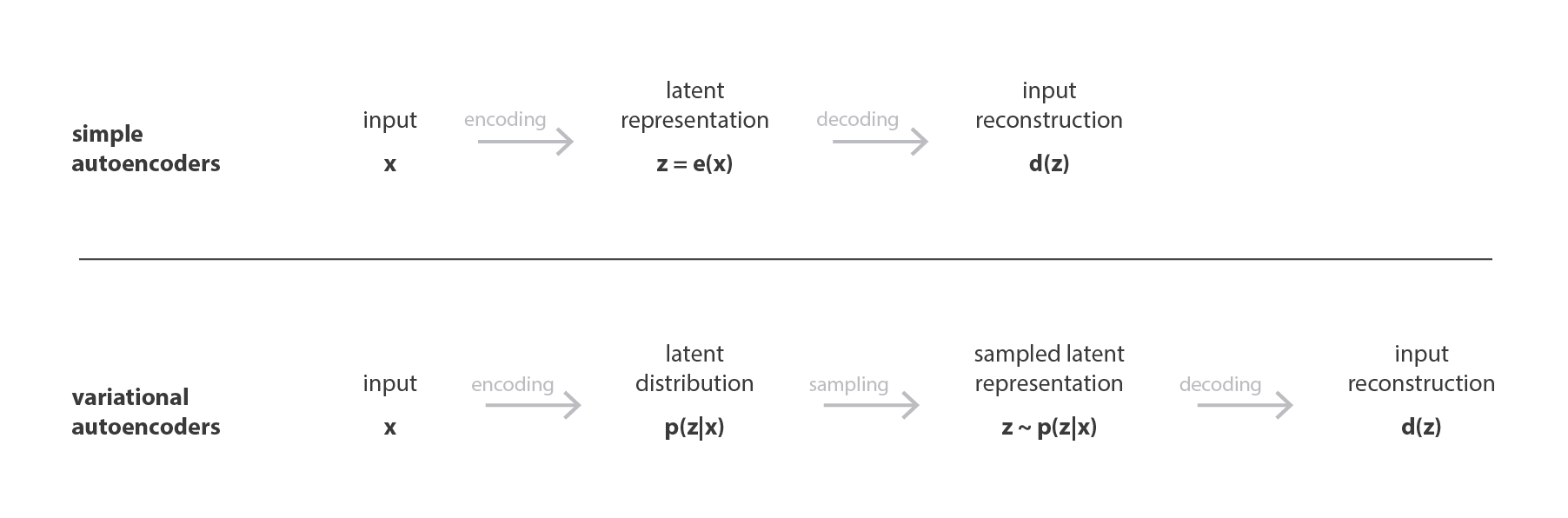
The general idea of autoencoders is pretty simple and consists in setting an encoder and a decoder as neural networks and to learn the best encoding-decoding scheme using an iterative optimisation process. So, at each iteration we feed the autoencoder architecture (the encoder followed by the decoder) with some data, we compare the encoded-decoded output with the initial data and backpropagate the error through the architecture to update the weights of the networks [3].



**Fig 3.2:** Autoencoder diagram and it’s loss function

Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimise the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularisation of the latent space, we proceed to a slight modification of the encoding-decoding process: instead of encoding an input as a single point, we encode it as a distribution over the latent space. The model is then trained as follows:

* The input is encoded as a distribution over the latent space
* A point from this space is sampled
* The sampled point is decoded and then reconstructed
* The reconstruction error is propagated through the network
* The process of sampling can be seen in figure 3.3.



**Figure 3.3:** Difference between an autoencoder and a variational autoencoder

The below figure shows how a variational autoencoder can be used to generate images. The probability distribution learnt by the encoder also encodes the semantics of an image. As shown below, food items and animals show homophily within themselves, that is, they lie nearer to each other but food items and animals themselves are far away from each other. This represents a latent representation of the dataset that also encodes the semantics of the images. This can be seen in figure 3.4.

A diagram of different colored squares

Description automatically generated

**Figure 3.4:** Variational autoencoder used in probabilistic diffusion model

**3.1.2.** **CLIP Encoder**

CLIP, or Contrastive Language-Image Pretraining, is a revolutionary framework developed by OpenAI for learning visual representations grounded in natural language. Unlike traditional approaches that rely solely on images for training, CLIP leverages large-scale datasets of image-text pairs to train a single model to understand both images and text simultaneously. By learning to associate images with their corresponding textual descriptions across diverse domains, CLIP can perform various tasks, such as image classification, object detection, and image generation, with remarkable accuracy and generalization. This multimodal understanding enables CLIP to interpret and generate content based on natural language prompts, opening up new possibilities for multimodal AI applications across a wide range of domains.

CLIP is used in the probabilistic diffusion model as follows:

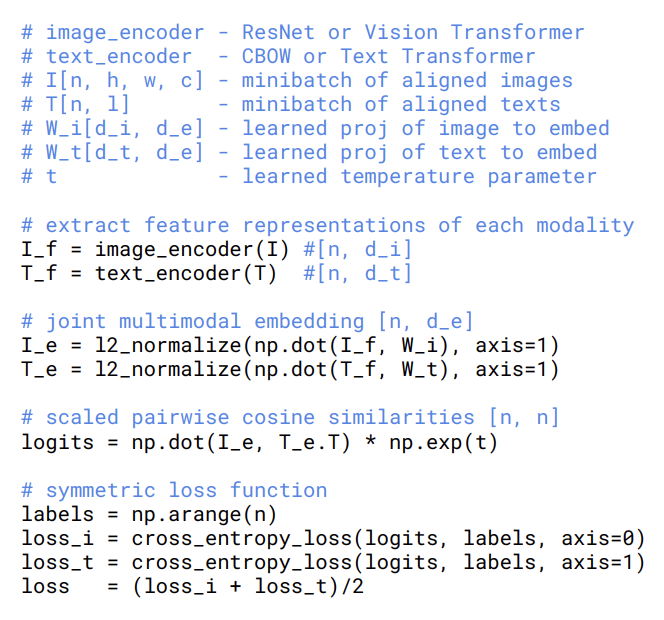
CLIP pre-trains an image encoder and a text encoder to predict which images were paired with which texts in our dataset. We then use this behavior to turn CLIP into a zero-shot classifier. We convert all of a dataset’s classes into captions such as “a photo of a dog” and predict the class of the caption CLIP estimates best pairs with a given image.

A diagram of a computer program

Description automatically generated

**Figure 3.5:** Contrastive Pre-training for CLIP [4]

CLIP serves as a guiding force for the image generation process. By providing textual prompts, users can influence the synthesis of images based on semantic descriptions. These prompts act as conditioning signals for the LDM, directing it to produce images that align with the semantics conveyed in the text.

****

**Figure 3.6:** Numpy-like pseudocode for the core implementation of CLIP [4]

* + 1. **U-Net**

U-Net is a convolutional neural network architecture designed for biomedical image segmentation tasks. It consists of a contracting path, which captures context through convolutional and pooling layers, followed by an expanding path that enables precise localization using upsampling and concatenation operations. Skip connections between corresponding layers in the contracting and expanding paths help retain spatial information, facilitating accurate segmentation. U-Net’s architecture enables it to handle challenges such as class imbalance and limited annotated data common in biomedical imaging, making it a powerful tool for tasks like cell and organ segmentation in medical diagnosis and research [2]. The U-Net architecture is shown in figure 3.7.

A diagram of a diagram

Description automatically generated with medium confidence

**Figure 3.7:** A Unet model that has downsampling and upsampling with resnet blocks.

* + 1. **DDPM (Denoising Diffusion Probabilistic Model)**

The DDPM (Denoising Diffusion Probabilistic Model) Scheduler is a component used in training denoising diffusion probabilistic models. Its primary purpose is to control the diffusion process during training. The diffusion process involves gradually adding noise to the input images and training the model to denoise them. The scheduler manages the schedule of noise levels, determining when and how much noise is added at each training step.

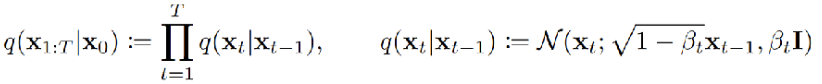
The key function of the DDPM Scheduler Is to adjust the diffusion process dynamically based on the training progress. It typically starts with low levels of noise and gradually increases the noise intensity as training progresses. This gradual increase helps the model learn to handle higher levels of noise and produce more accurate denoised images.

The scheduler’s parameters, such as the number of diffusion steps and the schedule of noise levels, are often optimized through experimentation to ensure efficient training and optimal performance of the denoising model. By controlling the diffusion process, the DDPM Scheduler plays a crucial role in training denoising diffusion probabilistic models effectively.[1]

The scheduler takes In the noise prediction outputted by the U-Net and removes the noise from the image. It has two processes:

1. **The forward process:**

The forward process is pretty simple. It adds noise to the training samples in a deterministic way.



**Figure 3.8:** Generating the noise for the next sample using DDPM [1]

As we can see, given that we know the noise for the previous image (which is of course in its latent space), the noise of the image for the next timestep can be calculated as a function of the previous timestep based on a Gaussian Distribution that depends solely on the previous image.

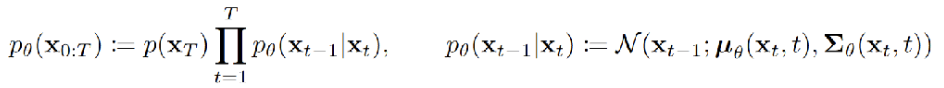
A deterministic formula exists, that allows us to calculate the noise to be provided at timestep t given that we know the noise at some previous stage (in this case, the initial stage where no noise was applied).



**Figure 3.9:** Generating the noise for sample at time t [1].

1. **The reverse process**

The reverse process involves going from a noisy image to a less noisier image, that is, denoising it. No deterministic formula exists for this process and this is why we make use of a neural network to learn how to predict the noise for each timestep.

****

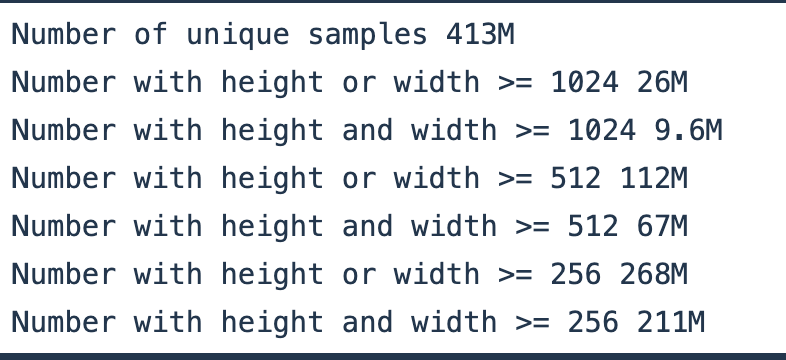
**Figure 3.10:** Noise removal for a given latent image by sampling from a Gaussian distribution. [1]

The Gaussian distribution is outputted by the U-Net and gives us the necessary mean and variance to go to an image on the previous timestamp.

* 1. **Data Collection and Augmentation**

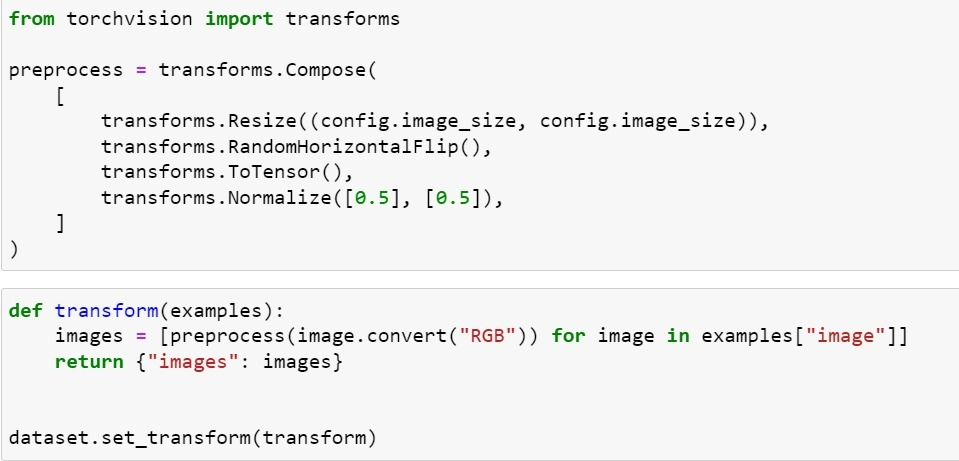
The dataset that’s used in the model is the LAION-5b dataset. It is a massive dataset with 5.85 billion pairs of images and tex. It’s got 2.3 billion pairs in English, 2.2 billion pairs in over 100 other languages, and 1 billion pairs where the language isn’t known. This dataset also comes with tools to help researchers explore it easily and find specific types of content. It was made by sorting through a ton of internet data, making sure it’s safe and useful for research. The good part about this is that it is open source and has thus been used as the dataset for the model we’re developing.

We’ll be using the English subset for this as the target audience for this project is users who are proficient in English. The English subset is consists of images with only English captions which is known as laion2B-en. A smaller subset of this known as laion-400m was used that uses 400 million images, and we restricted 400 million images to a height and width of atleast 512x512.. From the below image we can see that that would lead to a total of 67 million images. Out of these, 500,000 images were taken in random. This led to a total dataset size of 34 gb. The breakdown in shown in figure 3.11.

****

**Figure 3.11:** Distribution of image sizes in LAOIN-400M.

The images are still of different sizes though, so we’ll need to preprocess them first:



**Figure 3.12:** Image Preprocessing

* 1. **Training the model**

The most important parts of the model are the Unet and the DDPM scheduler. The job of the scheduler is to add / remove noise from the image. For example, here’s the output of the image that has noise added to it:



**Figure 3.13:** Sample image after adding noise to it.

The training config shown in figure 3.14:



**Figure 3.14:** Training Config for the model.

Training is done by adding noise to an image and iteratively removing it. During the denoising process the Unet learns the weights of the probability distribution. The number of epochs are 50 and the train\_batch\_size is of size 1000 so in each batch 500 images were taken. This is due to hardware constraints. Figure 3.15 shows the training epochs.

**A screenshot of a graph

Description automatically generated**

**Figure 3.15:** Last few epochs (out of 50) are shown.

**3.4. Model Evaluation**

Unlike standard machine learning models, FID – Frechet Inception Distance was used as the evaluation metric for the model evaluation.

* FID stands as a cornerstone metric that measures the distance between the distributions of generated and real images.
* Lower FID scores signify a closer match between generated and real-world images. In addition, it shows superior model performance in mimicking real data distributions.

**A graph of a diagram

Description automatically generated with medium confidence**

**Figure 3.16:** A measure of FID score of the model, with traditional models.

**A graph with a blue line

Description automatically generated**

**Figure 3.17:** Gradient scale vs IS

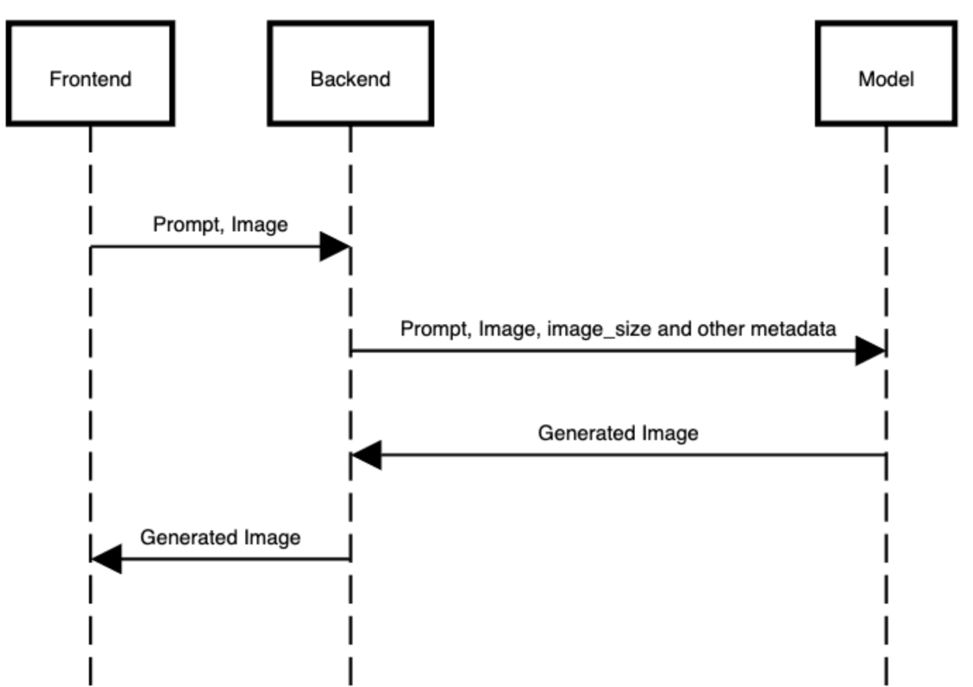
**A graph of a graph with red and blue lines

Description automatically generated**

**Figure 3.18:** Gradient Scale vs Precision, Recall

**3.5. Web App**

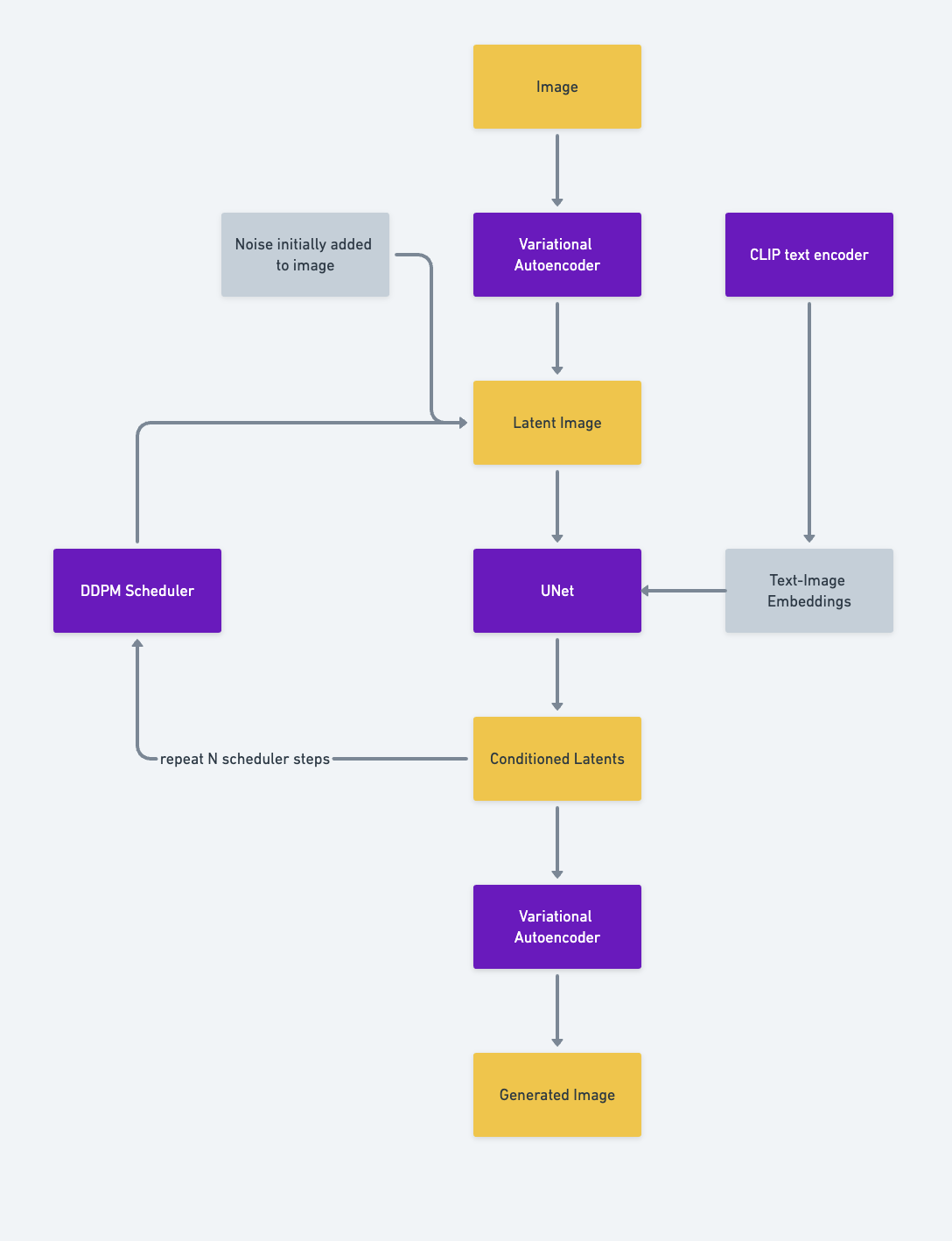
Finally, a simple web app is hooked up to ease the use of the model. It exposes fields for the prompts that is, the image and the text and allows you to generate variations from them. The interface for the web app has been shown in the results section. The webapp sends requests to the backend which is written in flask. Flask then uses the model and the relevant parameters as input and sends the request back the frontend for display.



**Figure 3.19:** Web App architecture

1. **Implementation**

The high level architecture of the model is as follows:

****

**Figure 4.1:** High level image-to-image latent diffuser architecture

The architecture consists of four major components whose implementation will be discussed in the following subsections.

**4.1 Variational Autoencoder**

The autoencoder consists of an encoder and a decoder. Both of these will be discussed in the subsequent subsections.

* + 1. **Encoder**

The encoder related logic can be found in the **encoder.py** file. This is responsible for converted the image from the dataset into its latent representation.

**4.1.1.1. VAE\_Encoder**: This class defines the encoder part of the Variational Autoencoder(VAE) architecture. It takes an input image and maps it to the latent space distribution parameters, mean and log variance. Here’s a breakdown:

* **\_\_init\_\_:** Initializes the encoder with a sequence of convolutional layers and residual blocks. It includes downsampling layers to reduce the spatial dimensions of the input.
* **forward**: Takes an input tensor `x` (image) and a noise tensor `noise`, and performs the following operations:
  + Iterates through each module in the encoder sequentially.
  + If the current module is a downsampling layer, asymmetric padding is applied to maintain the spatial dimensions correctly after downsampling.
  + Passes the input tensor through the layers.
  + Splits the output tensor into mean and log variance tensors.
  + Clamps the log variance within a certain range for numerical stability.
  + Computes the variance and standard deviation from the log variance.
  + Samples from the Gaussian distribution using the mean and standard deviation, scaled by the noise tensor.
  + Scales the sampled tensor by a constant factor.
  + Returns the encoded tensor.

This encoder takes an input image, processes it through convolutional and residual layers to extract features, and outputs the mean and log variance of the latent distribution. It also incorporates noise during encoding, which is crucial for the VAE to generate diverse and realistic samples during decoding.

* + 1. **Decoder**

Decoder related logic can be found in **decoder.py.** This file handles upsampling of the generated latent image into its normal representation.

**4.1.2.1. VAE\_AttentionBlock**: This class defines a block that incorporates self-attention mechanism into a Variational Autoencoder (VAE). Here’s a breakdown of its components:

**4.1.2.2. VAE\_ResidualBlock**: This class defines a residual block used in the VAE architecture. Residual blocks are used to facilitate the flow of gradients during training and help alleviate the vanishing gradient problem. Here’s a breakdown:

**4.1.2.3. VAE\_Decoder**: This class defines the decoder part of the VAE architecture. It’s a sequence of convolutional layers and residual blocks that upsamples the input to reconstruct the original image. Here’s a breakdown:

* **\_\_init\_\_**: Initializes the decoder with a sequence of convolutional layers and residual blocks. It includes upsampling layers to increase the spatial dimensions of the input.
* **forward**: Takes an input tensor `x` and performs the following operations:
  + Removes the scaling added by the encoder.
  + Passes the input tensor through the layers sequentially.
  + Returns the reconstructed image tensor.
  1. **CLIP Encoder**

This part of the model is responsible for converting text embeddings into image embeddings so that the text can act as guidance during the denoising process. Let’s break down the classes and methods in the provided code:

**4.2.1. CLIPEmbedding**

**1. Initialization:**

- **\_\_init\_\_(self, n\_vocab: int, n\_embd: int, n\_token: int)**: Initializes the embedding layer for tokens and a learnable parameter for position embeddings.

- **n\_vocab**: Number of tokens in the vocabulary.

- **n\_embd**: Dimensionality of token embeddings.

- **n\_token**: Number of tokens in the sequence.

**2. Forward Pass:**

- **forward(self, tokens)**: Performs the forward pass.

- **tokens**: Input token indices (Batch\_Size, Seq\_Len).

- Returns the embedded representation of tokens with positional encoding added (Batch\_Size, Seq\_Len, Dim).

**4.2.2. CLIPLayer**

**1. Initialization:**

- **\_\_init\_\_(self, n\_head: int, n\_embd: int)**: Initializes the CLIPLayer module.

- **n\_head**: Number of attention heads.

- **n\_embd**: Dimensionality of token embeddings.

**2. Forward Pass:**

- **forward(self, x)**: Performs the forward pass.

- **x**: Input tensor (Batch\_Size, Seq\_Len, Dim).

- Applies self-attention, residual connection, layer normalization, feedforward layer, activation function, another residual connection, and layer normalization.

- Returns the transformed tensor (Batch\_Size, Seq\_Len, Dim)

**4.2.3. CLIP**

**1. Initialization:**

- **\_\_init\_\_(self):** Initializes the CLIP model.

- Sets up the embedding layer and a stack of CLIPLayer modules.

**2. Forward Pass:**

- **forward(self, tokens: torch.LongTensor) -> torch.FloatTensor**: Performs the forward pass.

- **tokens**: Input token indices (Batch\_Size, Seq\_Len).

- Embeds the input tokens using the CLIPEmbedding module.

- Passes the embedded tokens through a stack of CLIPLayer modules.

- Applies layer normalization.

- Returns the final output tensor (Batch\_Size, Seq\_Len, Dim).

These classes define the components of the CLIP model, including token embedding, positional encoding, self-attention mechanism, and feedforward layers. The CLIP model is designed to process input tokens and output contextualized embeddings that capture semantic information from the input text.

1. **Unet**

The Unet is the most important component of the latent diffuser model as it removes noise for the scheduler iteratively. The various components used to build Unet are as follows:

* + 1. **TimeEmbedding:**

Performs linear transformations and activation functions on the input time embedding.

* + 1. **UNET\_ResidualBlock:**

Encapsulates a residual block within the Unet architecture, including group normalization, convolution, and residual connections.

* + 1. **UNET\_AttentionBlock:**

Represents an attention block within the Unet architecture, incorporating self-attention, cross-attention, and GeGLU activations.

* + 1. **Upsample:**

Implements upsampling of feature maps using nearest-neighbor interpolation followed by convolution.

* + 1. **SwitchSequential:**

Sequential module with specialized handling of different types of layers during forward pass based on their types.

* + 1. **UNET:**

Unet architecture composed of encoder, bottleneck, and decoder components for image-to-image translation tasks.

**1. Initialization:**

- **\_\_init\_\_(self)**: Initializes the UNET module with encoder, bottleneck, and decoder components.

**2. Forward Pass:**

- **forward(self, x, context, time)**: Executes the forward pass.

- **x**: Input feature tensor.

- **context**: Context tensor.

- **time**: Time embedding tensor.

- Passes the input through the encoder, bottleneck, and decoder sections of the UNET architecture.

- Returns the output tensor.

* + 1. **UNET\_OutputLayer:**

Output layer of the Unet architecture, applying group normalization, activation, and convolution.

* + 1. **Diffusion:**

Combines time embedding, Unet architecture, and output layer for image diffusion tasks.

* 1. **DDPM**

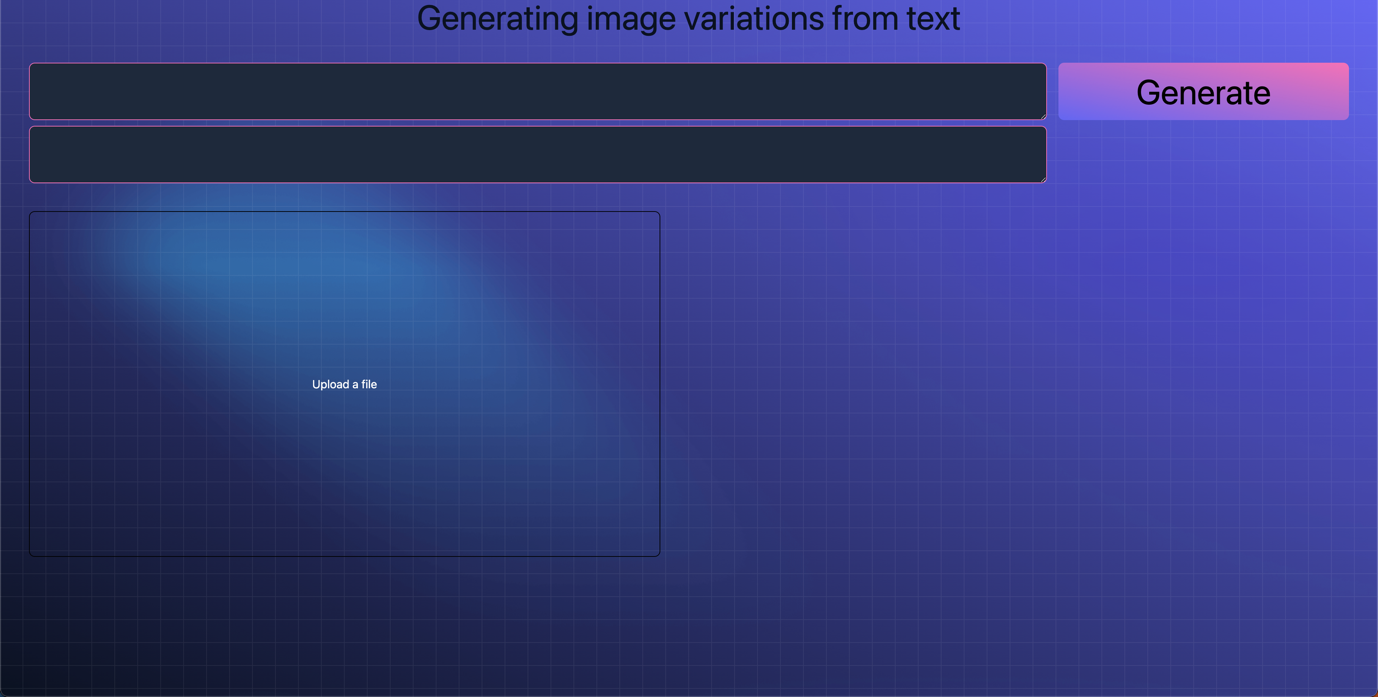
This **DDPMSampler** class implements sampling techniques for Diffusion Probabilistic Models (DDPM). Table 2.2 gives an overview of the DDPM class.

|  |  |
| --- | --- |
| Method Name | Description |
| **\_\_init\_\_** | Initializes the sampler with parameters such as the number of training steps, starting and ending beta values, and a random number generator. |
| **Set\_inference\_timesteps** | Sets the number of inference steps for the sampling process. |
| **\_get\_previous\_timestep** | Computes the previous timestep based on the current timestep for sampling. |
| **\_get\_variance** | Computes the variance for adding noise to the input based on the current timestep. |
| **Set\_strength** | Sets the strength of noise to add to the input image, affecting the distance between the output and the input. |
| **Step** | Performs a single step of sampling by computing alphas, betas, predicting original samples, computing coefficients, adding noise, and returning the sampled previous sample. |
| **Add\_noise** | Adds noise to original samples based on given timesteps. |

Table 2.2: DDPM Overview

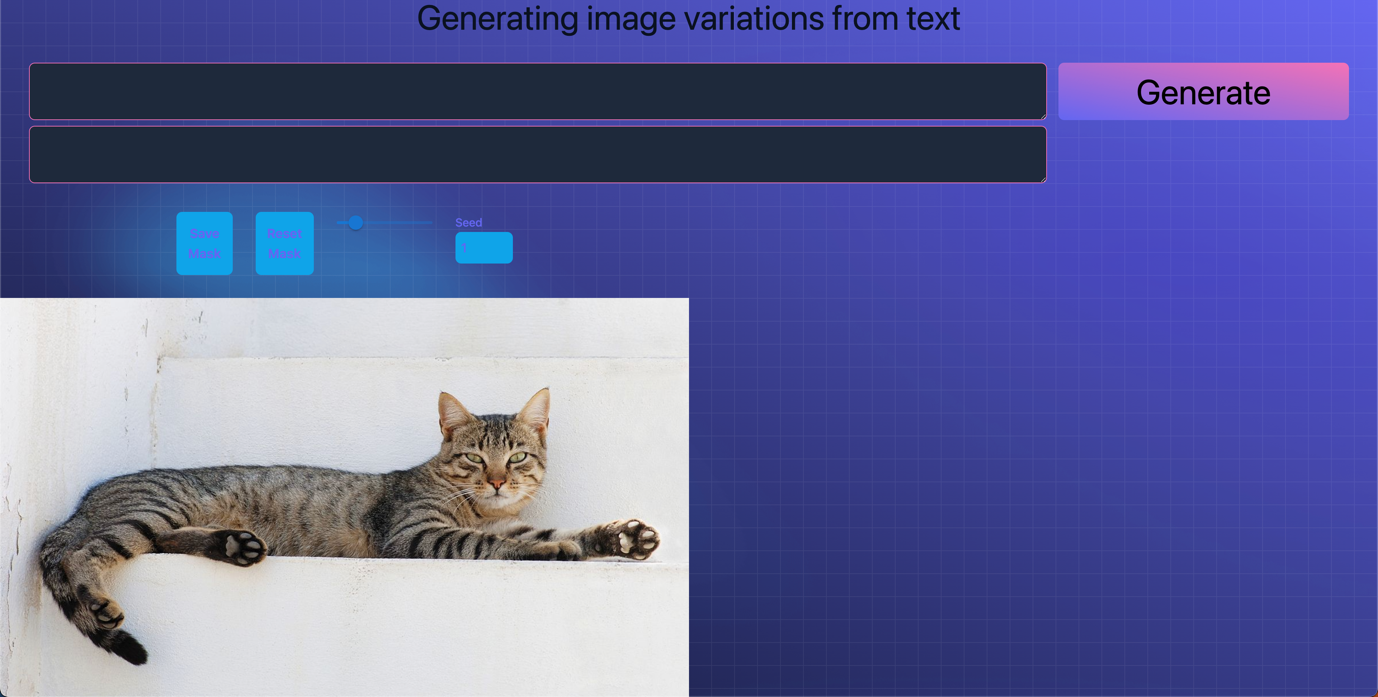
1. **Results**

The resulting model is made to be part of a broader web app. The app prompts the user to enter a prompt and to provide an image. The screen initially looks as follows:

****

**Figure 5.1**: Initial screen for the web app

On uploading an image, you get further options to edit it. The following screen is visible after uploading the image:

****

**Figure 5.2:** UI after uploading an image.

Let’s say we want the cat to be transformed into a tree! We add the **tree** prompt in the upper box and click on generate. We can a rendered image as follows:

**A screenshot of a computer

Description automatically generated**

**Figure 5.3:** Generated image.

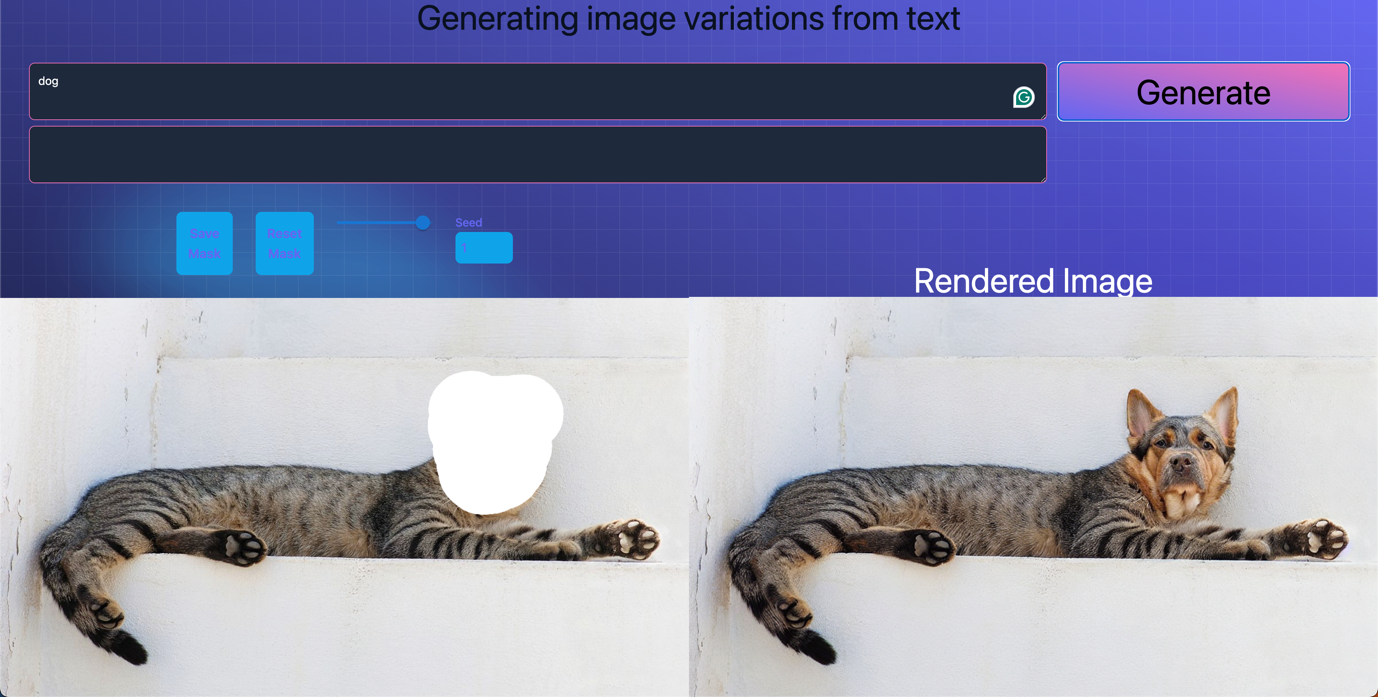
The app also allows you to edit specific parts of an image too. For example if we want to convert the head to a **dog**, we’ll make a mask on it.

**A cat lying on a ledge

Description automatically generated**

**Figure 5.4**: UI highlighting the masked parts of the images.

The result is a cat who’s **head** is now changed to a dog.

****

**Figure 5.5:** Generated image after applying the mask.

1. **Conclusion**

In conclusion, the project successfully leveraged latent diffusers for image-to-image (img2img) transformations and inpainting tasks, demonstrating the potential of these models in generating high-quality images from initial prompts and images.

The use of multiple models to create pipelines for iterative style transfer and content generation, showcased the versatility and creative possibilities of latent diffusion models. The project also highlighted the importance of optimizing memory and computational resources through techniques like model CPU offload and memory-efficient attention, ensuring efficient processing and real-time applications.

Furthermore, diffusion inpainting was also explored to be capable of regular editing tasks. This enables greater control during the image generation process which only takes a few seconds to process.

The development of both frontend and backend components in this project significantly enhances usability, making the application more accessible and efficient for users. The frontend, designed with user experience in mind, provides an intuitive interface for users to interact with the image-to-image transformations and inpainting functionalities. It allows users to easily upload images, specify prompts, and visualize the results in real-time, facilitating a seamless creative process.

The backend, on the other hand, is optimized for performance and scalability and handles all requests provided by the frontend. It is a simple wrapper around the API that exposes the interface of the model.

Together, the frontend and backend components work in harmony to create a powerful tool that not only meets the technical requirements of image processing tasks but also provides a user-friendly experience. This integration ensures that users can leverage the full potential of latent diffusers for creative applications, such as generating concept art, enhancing images, and more, all while maintaining a high level of usability and performance.

1. **References**
2. Jonathan Ho, Ajay Jain, Pieter Abbeel. Denoising Diffusion Probabilistic Models. Cornell University (arXiv) 2021.
3. Olaf Ronneberger, Philipp Fischer, Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv. 2015.
4. Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv. 2021
5. Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision. arXiv. 2021.
6. Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, Robin Rombach. SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis. arXiv. 2023
7. Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, Jun Zhu. DPM-Solver++: Fast Solver for Guided Sampling of Diffusion Probabilistic Models. arXiv. 2022

**Appendix**

**attention.py**

import torch

from torch import nn

from torch.nn import functional as F

import math

class SelfAttention(nn.Module):

def \_\_init\_\_(self, n\_heads, d\_embed, in\_proj\_bias=True, out\_proj\_bias=True):

super().\_\_init\_\_()

self.in\_proj = nn.Linear(d\_embed, 3 \* d\_embed, bias=in\_proj\_bias)

self.out\_proj = nn.Linear(d\_embed, d\_embed, bias=out\_proj\_bias)

self.n\_heads = n\_heads

self.d\_head = d\_embed // n\_heads

def forward(self, x, causal\_mask=False):

input\_shape = x.shape

batch\_size, sequence\_length, d\_embed = input\_shape

interim\_shape = (batch\_size, sequence\_length, self.n\_heads, self.d\_head)

q, k, v = self.in\_proj(x).chunk(3, dim=-1)

q = q.view(interim\_shape).transpose(1, 2)

k = k.view(interim\_shape).transpose(1, 2)

v = v.view(interim\_shape).transpose(1, 2)

weight = q @ k.transpose(-1, -2)

if causal\_mask:

mask = torch.ones\_like(weight, dtype=torch.bool).triu(1)

weight.masked\_fill\_(mask, -torch.inf)

weight /= math.sqrt(self.d\_head)

weight = F.softmax(weight, dim=-1)

output = weight @ v

output = output.transpose(1, 2)

output = output.reshape(input\_shape)

output = self.out\_proj(output)

return output

class CrossAttention(nn.Module):

def \_\_init\_\_(self, n\_heads, d\_embed, d\_cross, in\_proj\_bias=True, out\_proj\_bias=True):

super().\_\_init\_\_()

self.q\_proj = nn.Linear(d\_embed, d\_embed, bias=in\_proj\_bias)

self.k\_proj = nn.Linear(d\_cross, d\_embed, bias=in\_proj\_bias)

self.v\_proj = nn.Linear(d\_cross, d\_embed, bias=in\_proj\_bias)

self.out\_proj = nn.Linear(d\_embed, d\_embed, bias=out\_proj\_bias)

self.n\_heads = n\_heads

self.d\_head = d\_embed // n\_heads

def forward(self, x, y):

input\_shape = x.shape

batch\_size, sequence\_length, d\_embed = input\_shape

interim\_shape = (batch\_size, -1, self.n\_heads, self.d\_head)

q = self.q\_proj(x)

k = self.k\_proj(y)

v = self.v\_proj(y)

q = q.view(interim\_shape).transpose(1, 2)

k = k.view(interim\_shape).transpose(1, 2)

v = v.view(interim\_shape).transpose(1, 2)

weight = q @ k.transpose(-1, -2)

weight /= math.sqrt(self.d\_head)

weight = F.softmax(weight, dim=-1)

output = weight @ v

output = output.transpose(1, 2).contiguous()

output = output.view(input\_shape)

output = self.out\_proj(output)

return output

**clip.py**

import torch

from torch import nn

from torch.nn import functional as F

from attention import SelfAttention

class CLIPEmbedding(nn.Module):

def \_\_init\_\_(self, n\_vocab: int, n\_embd: int, n\_token: int):

super().\_\_init\_\_()

self.token\_embedding = nn.Embedding(n\_vocab, n\_embd)

self.position\_embedding = nn.Parameter(torch.zeros((n\_token, n\_embd)))

def forward(self, tokens):

x = self.token\_embedding(tokens)

x += self.position\_embedding

return x

class CLIPLayer(nn.Module):

def \_\_init\_\_(self, n\_head: int, n\_embd: int):

super().\_\_init\_\_()

self.layernorm\_1 = nn.LayerNorm(n\_embd)

self.attention = SelfAttention(n\_head, n\_embd)

self.layernorm\_2 = nn.LayerNorm(n\_embd)

self.linear\_1 = nn.Linear(n\_embd, 4 \* n\_embd)

self.linear\_2 = nn.Linear(4 \* n\_embd, n\_embd)

def forward(self, x):

residue = x

x = self.layernorm\_1(x)

x = self.attention(x, causal\_mask=True)

x += residue

residue = x

x = self.layernorm\_2(x)

x = self.linear\_1(x)

x = x \* torch.sigmoid(1.702 \* x)

x = self.linear\_2(x)

x += residue

return x

class CLIP(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.embedding = CLIPEmbedding(49408, 768, 77)

self.layers = nn.ModuleList([

CLIPLayer(12, 768) for i in range(12)

])

self.layernorm = nn.LayerNorm(768)

def forward(self, tokens: torch.LongTensor) -> torch.FloatTensor:

tokens = tokens.type(torch.long)

state = self.embedding(tokens)

for layer in self.layers:

state = layer(state)

output = self.layernorm(state)

return output

**ddpm.py**

import torch

import numpy as np

class DDPMSampler:

def \_\_init\_\_(self, generator: torch.Generator, num\_training\_steps=1000, beta\_start: float = 0.00085, beta\_end: float = 0.0120):

self.betas = torch.linspace(beta\_start \*\* 0.5, beta\_end \*\* 0.5, num\_training\_steps, dtype=torch.float32) \*\* 2

self.alphas = 1.0 - self.betas

self.alphas\_cumprod = torch.cumprod(self.alphas, dim=0)

self.one = torch.tensor(1.0)

self.generator = generator

self.num\_train\_timesteps = num\_training\_steps

self.timesteps = torch.from\_numpy(np.arange(0, num\_training\_steps)[::-1].copy())

def set\_inference\_timesteps(self, num\_inference\_steps=50):

self.num\_inference\_steps = num\_inference\_steps

step\_ratio = self.num\_train\_timesteps // self.num\_inference\_steps

timesteps = (np.arange(0, num\_inference\_steps) \* step\_ratio).round()[::-1].copy().astype(np.int64)

self.timesteps = torch.from\_numpy(timesteps)

def \_get\_previous\_timestep(self, timestep: int) -> int:

prev\_t = timestep - self.num\_train\_timesteps // self.num\_inference\_steps

return prev\_t

def \_get\_variance(self, timestep: int) -> torch.Tensor:

prev\_t = self.\_get\_previous\_timestep(timestep)

alpha\_prod\_t = self.alphas\_cumprod[timestep]

alpha\_prod\_t\_prev = self.alphas\_cumprod[prev\_t] if prev\_t >= 0 else self.one

current\_beta\_t = 1 - alpha\_prod\_t / alpha\_prod\_t\_prev

variance = (1 - alpha\_prod\_t\_prev) / (1 - alpha\_prod\_t) \* current\_beta\_t

variance = torch.clamp(variance, min=1e-20)

return variance

def set\_strength(self, strength=1):

start\_step = self.num\_inference\_steps - int(self.num\_inference\_steps \* strength)

self.timesteps = self.timesteps[start\_step:]

self.start\_step = start\_step

def step(self, timestep: int, latents: torch.Tensor, model\_output: torch.Tensor):

t = timestep

prev\_t = self.\_get\_previous\_timestep(t)

alpha\_prod\_t = self.alphas\_cumprod[t]

alpha\_prod\_t\_prev = self.alphas\_cumprod[prev\_t] if prev\_t >= 0 else self.one

beta\_prod\_t = 1 - alpha\_prod\_t

beta\_prod\_t\_prev = 1 - alpha\_prod\_t\_prev

current\_alpha\_t = alpha\_prod\_t / alpha\_prod\_t\_prev

current\_beta\_t = 1 - current\_alpha\_t

pred\_original\_sample = (latents - beta\_prod\_t \*\* (0.5) \* model\_output) / alpha\_prod\_t \*\* (0.5)

pred\_original\_sample\_coeff = (alpha\_prod\_t\_prev \*\* (0.5) \* current\_beta\_t) / beta\_prod\_t

current\_sample\_coeff = current\_alpha\_t \*\* (0.5) \* beta\_prod\_t\_prev / beta\_prod\_t

pred\_prev\_sample = pred\_original\_sample\_coeff \* pred\_original\_sample + current\_sample\_coeff \* latents

variance = 0

if t > 0:

device = model\_output.device

noise = torch.randn(model\_output.shape, generator=self.generator, device=device, dtype=model\_output.dtype)

variance = (self.\_get\_variance(t) \*\* 0.5) \* noise

pred\_prev\_sample = pred\_prev\_sample + variance

return pred\_prev\_sample

def add\_noise(

self,

original\_samples: torch.FloatTensor,

timesteps: torch.IntTensor,

) -> torch.FloatTensor:

alphas\_cumprod = self.alphas\_cumprod.to(device=original\_samples.device, dtype=original\_samples.dtype)

timesteps = timesteps.to(original\_samples.device)

sqrt\_alpha\_prod = alphas\_cumprod[timesteps] \*\* 0.5

sqrt\_alpha\_prod = sqrt\_alpha\_prod.flatten()

while len(sqrt\_alpha\_prod.shape) < len(original\_samples.shape):

sqrt\_alpha\_prod = sqrt\_alpha\_prod.unsqueeze(-1)

sqrt\_one\_minus\_alpha\_prod = (1 - alphas\_cumprod[timesteps]) \*\* 0.5

sqrt\_one\_minus\_alpha\_prod = sqrt\_one\_minus\_alpha\_prod.flatten()

while len(sqrt\_one\_minus\_alpha\_prod.shape) < len(original\_samples.shape):

sqrt\_one\_minus\_alpha\_prod = sqrt\_one\_minus\_alpha\_prod.unsqueeze(-1)

noise = torch.randn(original\_samples.shape, generator=self.generator, device=original\_samples.device, dtype=original\_samples.dtype)

noisy\_samples = sqrt\_alpha\_prod \* original\_samples + sqrt\_one\_minus\_alpha\_prod \* noise

return noisy\_samples

**decoder.py**

import torch

from torch import nn

from torch.nn import functional as F

from attention import SelfAttention

class VAE\_AttentionBlock(nn.Module):

def \_\_init\_\_(self, channels):

super().\_\_init\_\_()

self.groupnorm = nn.GroupNorm(32, channels)

self.attention = SelfAttention(1, channels)

def forward(self, x):

residue = x

x = self.groupnorm(x)

n, c, h, w = x.shape

x = x.view((n, c, h \* w))

x = x.transpose(-1, -2)

x = self.attention(x)

x = x.transpose(-1, -2)

x = x.view((n, c, h, w))

x += residue

return x

class VAE\_ResidualBlock(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super().\_\_init\_\_()

self.groupnorm\_1 = nn.GroupNorm(32, in\_channels)

self.conv\_1 = nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1)

self.groupnorm\_2 = nn.GroupNorm(32, out\_channels)

self.conv\_2 = nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1)

if in\_channels == out\_channels:

self.residual\_layer = nn.Identity()

else:

self.residual\_layer = nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, padding=0)

def forward(self, x):

residue = x

x = self.groupnorm\_1(x)

x = F.silu(x)

x = self.conv\_1(x)

x = self.groupnorm\_2(x)

x = F.silu(x)

x = self.conv\_2(x)

return x + self.residual\_layer(residue)

class VAE\_Decoder(nn.Sequential):

def \_\_init\_\_(self):

super().\_\_init\_\_(

nn.Conv2d(4, 4, kernel\_size=1, padding=0),

nn.Conv2d(4, 512, kernel\_size=3, padding=1),

VAE\_ResidualBlock(512, 512),

VAE\_AttentionBlock(512),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

nn.Upsample(scale\_factor=2),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

nn.Upsample(scale\_factor=2),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

VAE\_ResidualBlock(512, 256),

VAE\_ResidualBlock(256, 256),

VAE\_ResidualBlock(256, 256),

nn.Upsample(scale\_factor=2),

nn.Conv2d(256, 256, kernel\_size=3, padding=1),

VAE\_ResidualBlock(256, 128),

VAE\_ResidualBlock(128, 128),

VAE\_ResidualBlock(128, 128),

nn.GroupNorm(32, 128),

nn.SiLU(),

nn.Conv2d(128, 3, kernel\_size=3, padding=1),

)

def forward(self, x):

x /= 0.18215

for module in self:

x = module(x)

return x

**diffusion.py**

import torch

from torch import nn

from torch.nn import functional as F

from attention import SelfAttention, CrossAttention

class TimeEmbedding(nn.Module):

def \_\_init\_\_(self, n\_embd):

super().\_\_init\_\_()

self.linear\_1 = nn.Linear(n\_embd, 4 \* n\_embd)

self.linear\_2 = nn.Linear(4 \* n\_embd, 4 \* n\_embd)

def forward(self, x):

x = self.linear\_1(x)

x = F.silu(x)

x = self.linear\_2(x)

return x

class UNET\_ResidualBlock(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, n\_time=1280):

super().\_\_init\_\_()

self.groupnorm\_feature = nn.GroupNorm(32, in\_channels)

self.conv\_feature = nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1)

self.linear\_time = nn.Linear(n\_time, out\_channels)

self.groupnorm\_merged = nn.GroupNorm(32, out\_channels)

self.conv\_merged = nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1)

if in\_channels == out\_channels:

self.residual\_layer = nn.Identity()

else:

self.residual\_layer = nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, padding=0)

def forward(self, feature, time):

residue = feature

feature = self.groupnorm\_feature(feature)

feature = F.silu(feature)

feature = self.conv\_feature(feature)

time = F.silu(time)

time = self.linear\_time(time)

merged = feature + time.unsqueeze(-1).unsqueeze(-1)

merged = self.groupnorm\_merged(merged)

merged = F.silu(merged)

merged = self.conv\_merged(merged)

return merged + self.residual\_layer(residue)

class UNET\_AttentionBlock(nn.Module):

def \_\_init\_\_(self, n\_head: int, n\_embd: int, d\_context=768):

super().\_\_init\_\_()

channels = n\_head \* n\_embd

self.groupnorm = nn.GroupNorm(32, channels, eps=1e-6)

self.conv\_input = nn.Conv2d(channels, channels, kernel\_size=1, padding=0)

self.layernorm\_1 = nn.LayerNorm(channels)

self.attention\_1 = SelfAttention(n\_head, channels, in\_proj\_bias=False)

self.layernorm\_2 = nn.LayerNorm(channels)

self.attention\_2 = CrossAttention(n\_head, channels, d\_context, in\_proj\_bias=False)

self.layernorm\_3 = nn.LayerNorm(channels)

self.linear\_geglu\_1 = nn.Linear(channels, 4 \* channels \* 2)

self.linear\_geglu\_2 = nn.Linear(4 \* channels, channels)

self.conv\_output = nn.Conv2d(channels, channels, kernel\_size=1, padding=0)

def forward(self, x, context):

residue\_long = x

x = self.groupnorm(x)

x = self.conv\_input(x)

n, c, h, w = x.shape

x = x.view((n, c, h \* w))

x = x.transpose(-1, -2)

residue\_short = x

x = self.layernorm\_1(x)

x = self.attention\_1(x)

x += residue\_short

residue\_short = x

x = self.layernorm\_2(x)

x = self.attention\_2(x, context)

x += residue\_short

residue\_short = x

x = self.layernorm\_3(x)

x, gate = self.linear\_geglu\_1(x).chunk(2, dim=-1)

x = x \* F.gelu(gate)

x = self.linear\_geglu\_2(x)

x += residue\_short

x = x.transpose(-1, -2)

x = x.view((n, c, h, w))

return self.conv\_output(x) + residue\_long

class Upsample(nn.Module):

def \_\_init\_\_(self, channels):

super().\_\_init\_\_()

self.conv = nn.Conv2d(channels, channels, kernel\_size=3, padding=1)

def forward(self, x):

x = F.interpolate(x, scale\_factor=2, mode='nearest')

return self.conv(x)

class SwitchSequential(nn.Sequential):

def forward(self, x, context, time):

for layer in self:

if isinstance(layer, UNET\_AttentionBlock):

x = layer(x, context)

elif isinstance(layer, UNET\_ResidualBlock):

x = layer(x, time)

else:

x = layer(x)

return x

class UNET(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.encoders = nn.ModuleList([

SwitchSequential(nn.Conv2d(4, 320, kernel\_size=3, padding=1)),

SwitchSequential(UNET\_ResidualBlock(320, 320), UNET\_AttentionBlock(8, 40)),

SwitchSequential(UNET\_ResidualBlock(320, 320), UNET\_AttentionBlock(8, 40)),

SwitchSequential(nn.Conv2d(320, 320, kernel\_size=3, stride=2, padding=1)),

SwitchSequential(UNET\_ResidualBlock(320, 640), UNET\_AttentionBlock(8, 80)),

SwitchSequential(UNET\_ResidualBlock(640, 640), UNET\_AttentionBlock(8, 80)),

SwitchSequential(nn.Conv2d(640, 640, kernel\_size=3, stride=2, padding=1)),

SwitchSequential(UNET\_ResidualBlock(640, 1280), UNET\_AttentionBlock(8, 160)),

SwitchSequential(UNET\_ResidualBlock(1280, 1280), UNET\_AttentionBlock(8, 160)),

SwitchSequential(nn.Conv2d(1280, 1280, kernel\_size=3, stride=2, padding=1)),

SwitchSequential(UNET\_ResidualBlock(1280, 1280)),

SwitchSequential(UNET\_ResidualBlock(1280, 1280)),

])

self.bottleneck = SwitchSequential(

UNET\_ResidualBlock(1280, 1280),

UNET\_AttentionBlock(8, 160),

UNET\_ResidualBlock(1280, 1280),

)

self.decoders = nn.ModuleList([

SwitchSequential(UNET\_ResidualBlock(2560, 1280)),

SwitchSequential(UNET\_ResidualBlock(2560, 1280)),

SwitchSequential(UNET\_ResidualBlock(2560, 1280), Upsample(1280)),

SwitchSequential(UNET\_ResidualBlock(2560, 1280), UNET\_AttentionBlock(8, 160)),

SwitchSequential(UNET\_ResidualBlock(2560, 1280), UNET\_AttentionBlock(8, 160)),

SwitchSequential(UNET\_ResidualBlock(1920, 1280), UNET\_AttentionBlock(8, 160), Upsample(1280)),

SwitchSequential(UNET\_ResidualBlock(1920, 640), UNET\_AttentionBlock(8, 80)),

SwitchSequential(UNET\_ResidualBlock(1280, 640), UNET\_AttentionBlock(8, 80)),

SwitchSequential(UNET\_ResidualBlock(960, 640), UNET\_AttentionBlock(8, 80), Upsample(640)),

SwitchSequential(UNET\_ResidualBlock(960, 320), UNET\_AttentionBlock(8, 40)),

SwitchSequential(UNET\_ResidualBlock(640, 320), UNET\_AttentionBlock(8, 40)),

SwitchSequential(UNET\_ResidualBlock(640, 320), UNET\_AttentionBlock(8, 40)),

])

def forward(self, x, context, time):

skip\_connections = []

for layers in self.encoders:

x = layers(x, context, time)

skip\_connections.append(x)

x = self.bottleneck(x, context, time)

for layers in self.decoders:

x = torch.cat((x, skip\_connections.pop()), dim=1)

x = layers(x, context, time)

return x

class UNET\_OutputLayer(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super().\_\_init\_\_()

self.groupnorm = nn.GroupNorm(32, in\_channels)

self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1)

def forward(self, x):

x = self.groupnorm(x)

x = F.silu(x)

x = self.conv(x)

return x

class Diffusion(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.time\_embedding = TimeEmbedding(320)

self.unet = UNET()

self.final = UNET\_OutputLayer(320, 4)

def forward(self, latent, context, time):

time = self.time\_embedding(time)

output = self.unet(latent, context, time)

output = self.final(output)

return output

**encoder.py**

import torch

from torch import nn

from torch.nn import functional as F

from decoder import VAE\_AttentionBlock, VAE\_ResidualBlock

class VAE\_Encoder(nn.Sequential):

def \_\_init\_\_(self):

super().\_\_init\_\_(

nn.Conv2d(3, 128, kernel\_size=3, padding=1),

VAE\_ResidualBlock(128, 128),

VAE\_ResidualBlock(128, 128),

nn.Conv2d(128, 128, kernel\_size=3, stride=2, padding=0),

VAE\_ResidualBlock(128, 256),

VAE\_ResidualBlock(256, 256),

nn.Conv2d(256, 256, kernel\_size=3, stride=2, padding=0),

VAE\_ResidualBlock(256, 512),

VAE\_ResidualBlock(512, 512),

nn.Conv2d(512, 512, kernel\_size=3, stride=2, padding=0),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

VAE\_ResidualBlock(512, 512),

VAE\_AttentionBlock(512),

VAE\_ResidualBlock(512, 512),

nn.GroupNorm(32, 512),

nn.SiLU(),

nn.Conv2d(512, 8, kernel\_size=3, padding=1),

nn.Conv2d(8, 8, kernel\_size=1, padding=0),

)

def forward(self, x, noise):

for module in self:

if getattr(module, 'stride', None) == (2, 2):

x = F.pad(x, (0, 1, 0, 1))

x = module(x)

mean, log\_variance = torch.chunk(x, 2, dim=1)

log\_variance = torch.clamp(log\_variance, -30, 20)

variance = log\_variance.exp()

stdev = variance.sqrt()

x = mean + stdev \* noise

x \*= 0.18215

return x

**pipeline.py**

import torch

import numpy as np

from tqdm import tqdm

from ddpm import DDPMSampler

WIDTH = 512

HEIGHT = 512

LATENTS\_WIDTH = WIDTH // 8

LATENTS\_HEIGHT = HEIGHT // 8

def generate(

prompt,

uncond\_prompt=None,

input\_image=None,

strength=0.8,

do\_cfg=True,

cfg\_scale=7.5,

sampler\_name="ddpm",

n\_inference\_steps=50,

models={},

seed=None,

device=None,

idle\_device=None,

tokenizer=None,

):

with torch.no\_grad():

if not 0 < strength <= 1:

raise ValueError("strength must be between 0 and 1")

if idle\_device:

to\_idle = lambda x: x.to(idle\_device)

else:

to\_idle = lambda x: x

# Initialize random number generator according to the seed specified

generator = torch.Generator(device=device)

if seed is None:

generator.seed()

else:

generator.manual\_seed(seed)

clip = models["clip"]

clip.to(device)

if do\_cfg:

cond\_tokens = tokenizer.batch\_encode\_plus(

[prompt], padding="max\_length", max\_length=77

).input\_ids

cond\_tokens = torch.tensor(cond\_tokens, dtype=torch.long, device=device)

cond\_context = clip(cond\_tokens)

uncond\_tokens = tokenizer.batch\_encode\_plus(

[uncond\_prompt], padding="max\_length", max\_length=77

).input\_ids

uncond\_tokens = torch.tensor(uncond\_tokens, dtype=torch.long, device=device)

uncond\_context = clip(uncond\_tokens)

context = torch.cat([cond\_context, uncond\_context])

else:

tokens = tokenizer.batch\_encode\_plus(

[prompt], padding="max\_length", max\_length=77

).input\_ids

tokens = torch.tensor(tokens, dtype=torch.long, device=device)

context = clip(tokens)

to\_idle(clip)

if sampler\_name == "ddpm":

sampler = DDPMSampler(generator)

sampler.set\_inference\_timesteps(n\_inference\_steps)

else:

raise ValueError("Unknown sampler value %s. ")

latents\_shape = (1, 4, LATENTS\_HEIGHT, LATENTS\_WIDTH)

if input\_image:

encoder = models["encoder"]

encoder.to(device)

input\_image\_tensor = input\_image.resize((WIDTH, HEIGHT))

input\_image\_tensor = np.array(input\_image\_tensor)

input\_image\_tensor = torch.tensor(input\_image\_tensor, dtype=torch.float32, device=device)

input\_image\_tensor = rescale(input\_image\_tensor, (0, 255), (-1, 1))

input\_image\_tensor = input\_image\_tensor.unsqueeze(0)

input\_image\_tensor = input\_image\_tensor.permute(0, 3, 1, 2)

encoder\_noise = torch.randn(latents\_shape, generator=generator, device=device)

latents = encoder(input\_image\_tensor, encoder\_noise)

sampler.set\_strength(strength=strength)

latents = sampler.add\_noise(latents, sampler.timesteps[0])

to\_idle(encoder)

else:

latents = torch.randn(latents\_shape, generator=generator, device=device)

diffusion = models["diffusion"]

diffusion.to(device)

timesteps = tqdm(sampler.timesteps)

for i, timestep in enumerate(timesteps):

time\_embedding = get\_time\_embedding(timestep).to(device)

model\_input = latents

if do\_cfg:

model\_input = model\_input.repeat(2, 1, 1, 1)

model\_output = diffusion(model\_input, context, time\_embedding)

if do\_cfg:

output\_cond, output\_uncond = model\_output.chunk(2)

model\_output = cfg\_scale \* (output\_cond - output\_uncond) + output\_uncond

latents = sampler.step(timestep, latents, model\_output)

to\_idle(diffusion)

decoder = models["decoder"]

decoder.to(device)

images = decoder(latents)

to\_idle(decoder)

images = rescale(images, (-1, 1), (0, 255), clamp=True)

images = images.permute(0, 2, 3, 1)

images = images.to("cpu", torch.uint8).numpy()

return images[0]

def rescale(x, old\_range, new\_range, clamp=False):

old\_min, old\_max = old\_range

new\_min, new\_max = new\_range

x -= old\_min

x \*= (new\_max - new\_min) / (old\_max - old\_min)

x += new\_min

if clamp:

x = x.clamp(new\_min, new\_max)

return x

def get\_time\_embedding(timestep):

# Shape: (160,)

freqs = torch.pow(10000, -torch.arange(start=0, end=160, dtype=torch.float32) / 160)

x = torch.tensor([timestep], dtype=torch.float32)[:, None] \* freqs[None]

return torch.cat([torch.cos(x), torch.sin(x)], dim=-1)