Diffusion Model API: A Full-Stack ML Deployment Project

Abrar Zahin

This project demonstrates the end-to-end deployment of an image generation service using a diffusion model. It uses the pre-trained Stable Diffusion model `CompVis/stable-diffusion-v1-4` from Hugging Face to convert text prompts into realistic images.

Model Description: This is a model that can be used to generate and modify images based on text prompts. It is a Latent Diffusion Model that uses a fixed, pretrained text encoder (CLIP ViT-L/14) as suggested in the Imagen paper. Following text blocks are directly copied from the above link for learning purpose:

Generally speaking, diffusion models are machine learning systems that are trained to denoise random Gaussian noise step by step, to get to a sample of interest, such as an image.

Diffusion models have shown to achieve state-of-the-art results for generating image data. But one downside of diffusion models is that the reverse denoising process is slow because of its repeated, sequential nature. In addition, these models consume a lot of memory because they operate in pixel space, which becomes huge when generating high-resolution images. Therefore, it is challenging to train these models and also use them for inference.

Latent diffusion can reduce the memory and compute complexity by applying the diffusion process over a lower dimensional latent space, instead of using the actual pixel space. **This is the key difference between standard diffusion and latent diffusion models: in latent diffusion the model is trained to generate latent (compressed) representations of the images.**

There are three main components in latent diffusion.

1. **An autoencoder (VAE).** The VAE model has two parts, an encoder and a decoder. The encoder is used to convert the image into a low dimensional latent representation, which will serve as the input to the U-Net model. The decoder, conversely, transforms the latent representation back into an image.

   During latent diffusion training, the encoder is used to get the latent representations (latents) of the images for the forward diffusion process, which applies more and more noise at each step. During inference, the denoised latents generated by the reverse diffusion process are converted back into images using the VAE decoder. As we will see during inference we only need the VAE decoder.

2. **A U-Net.** The U-Net has an encoder part and a decoder part both comprised of ResNet blocks. The encoder compresses an image representation into a lower resolution image representation and the decoder decodes the lower resolution image representation back to the original higher resolution image representation that is supposedly less noisy. More specifically, the U-Net output predicts the noise residual which can be used to compute the predicted denoised image representation.

   To prevent the U-Net from losing important information while downsampling, short-cut connections are usually added between the downsampling ResNets of the encoder to the upsampling ResNets of the decoder. Additionally, the stable diffusion U-Net is able to condition its output on text-embeddings via cross-attention layers. The cross-attention layers are added to both the encoder and decoder part of the U-Net usually between ResNet blocks.

3. **A text-encoder, e.g. CLIP's Text Encoder.** The text-encoder is responsible for transforming the input prompt, e.g. ”An astronaut riding a horse” into an embedding space that can be understood by the U-Net. It is usually a simple transformer-based encoder that maps a sequence of input tokens to a sequence of latent text-embeddings.

   Inspired by Imagen, Stable Diffusion does not train the text-encoder during training and simply uses an CLIP's already trained text encoder, CLIPTextModel.

Why is latent diffusion fast and efficient?

Since latent diffusion operates on a low dimensional space, it greatly reduces the memory and compute requirements compared to pixel-space diffusion models. For example, the autoencoder used in Stable Diffusion has a reduction factor of 8. This means that an image of shape (3, 512, 512) becomes (4, 64, 64) in latent space, which means the spatial compression ratio is $8 \times 8 = 64$.

This is why it's possible to generate 512×512 images so quickly, even on 16GB Colab GPUs!

The stable diffusion model takes both a latent seed and a text prompt as an input. The latent seed is then used to generate random latent image representations of size 6464 where as the text prompt is transformed to text embeddings of size 77768 via CLIP's text encoder.

Next the U-Net iteratively denoises the random latent image representations while being conditioned on the text embeddings. The output of the U-Net, being the noise residual, is used to compute a denoised latent image representation via a scheduler algorithm. Many different scheduler algorithms can be used for this computation, each having its pro- and cons. For Stable Diffusion, we recommend using one of:

PNDM scheduler (used by default) DDIM scheduler K-LMS scheduler Theory on how the scheduler algorithm function is out-of-scope for this notebook, but in short one should remember that they compute the predicted denoised image representation from the previous noise representation and the predicted noise residual. For more information, we recommend looking into Elucidating the Design Space of Diffusion-Based Generative Models

The denoising process is repeated ca. 50 times to step-by-step retrieve better latent image representations. Once complete, the latent image representation is decoded by the decoder part of the variational auto encoder.

1 Serving the Application:

The application is served via FastAPI, a modern and fast web framework for building APIs with Python. A single POST endpoint `/generate` accepts a JSON payload containing a text prompt, which is passed to the model. The output is a PNG image returned as a streaming HTTP response.

To ensure environment consistency and portability, the application is packaged using Docker. The Dockerfile sets up the environment, installs required packages like `diffusers`, `torch`, and `Pillow`, and starts the FastAPI server with Uvicorn.

For real-world deployment, Kubernetes manifests are included to create:

- A `Deployment` with two replicas of the containerized API
- A `Service` of type `LoadBalancer` for external access
- A `Horizontal Pod Autoscaler (HPA)` to automatically scale pods based on CPU usage

The system is designed to simulate production environments where large-scale, on-demand generation of images is required. While this version runs on limited hardware (e.g., Google Colab), the architecture is robust and scalable to high-performance cloud infrastructure with GPUs, supporting much larger models like 7B or 70B parameter LLMs.

Overall, the project demonstrates practical experience in machine learning model serving, API development, containerization with Docker, and orchestration with Kubernetes. It highlights the ability to move beyond model training to full production-grade deployment pipelines.

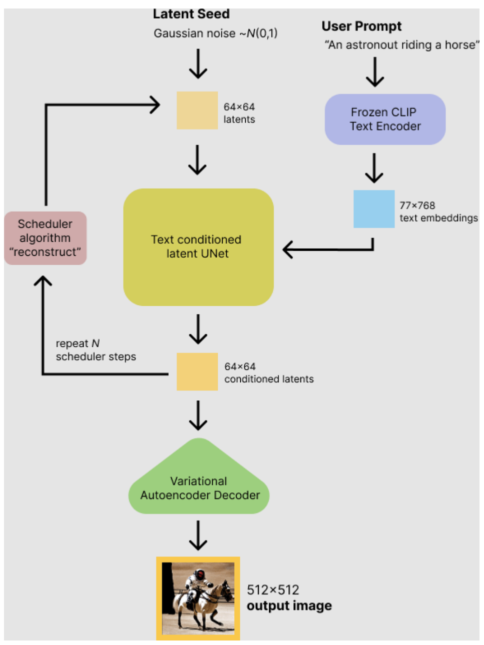

Figure 1: Diffusion model flow