How to implement Stable Diffusion

After seeing how Stable Diffusion works theoretically, now it's time to implement it in Python.

Artificial Intelligence | Stable Diffusion

In a previous article, we saw how Stable Diffusion works going into detail but without using a single line of code. We saw how a model is trained (forward diffusion) and then use it in the inference process to generate spectacular images with Artificial Intelligence (reverse diffusion). If you haven't read that article yet, I recommend you do so before continuing with this one.

In this article **we are going to implement each part of the inference process** using Python code, in order to understand how it works in a more technical way. You may not fully understand all the lines of code in this article. Don't worry, it doesn't matter. The idea is to see in general how it works, what its components are and how they interact with each other.

We are going to use the same version as in the previous article (Stable Diffusion 1.5) and a text-to-image (txt2img) conditioning.

Let's start by recalling the steps of the inference process using the final image from the previous article:

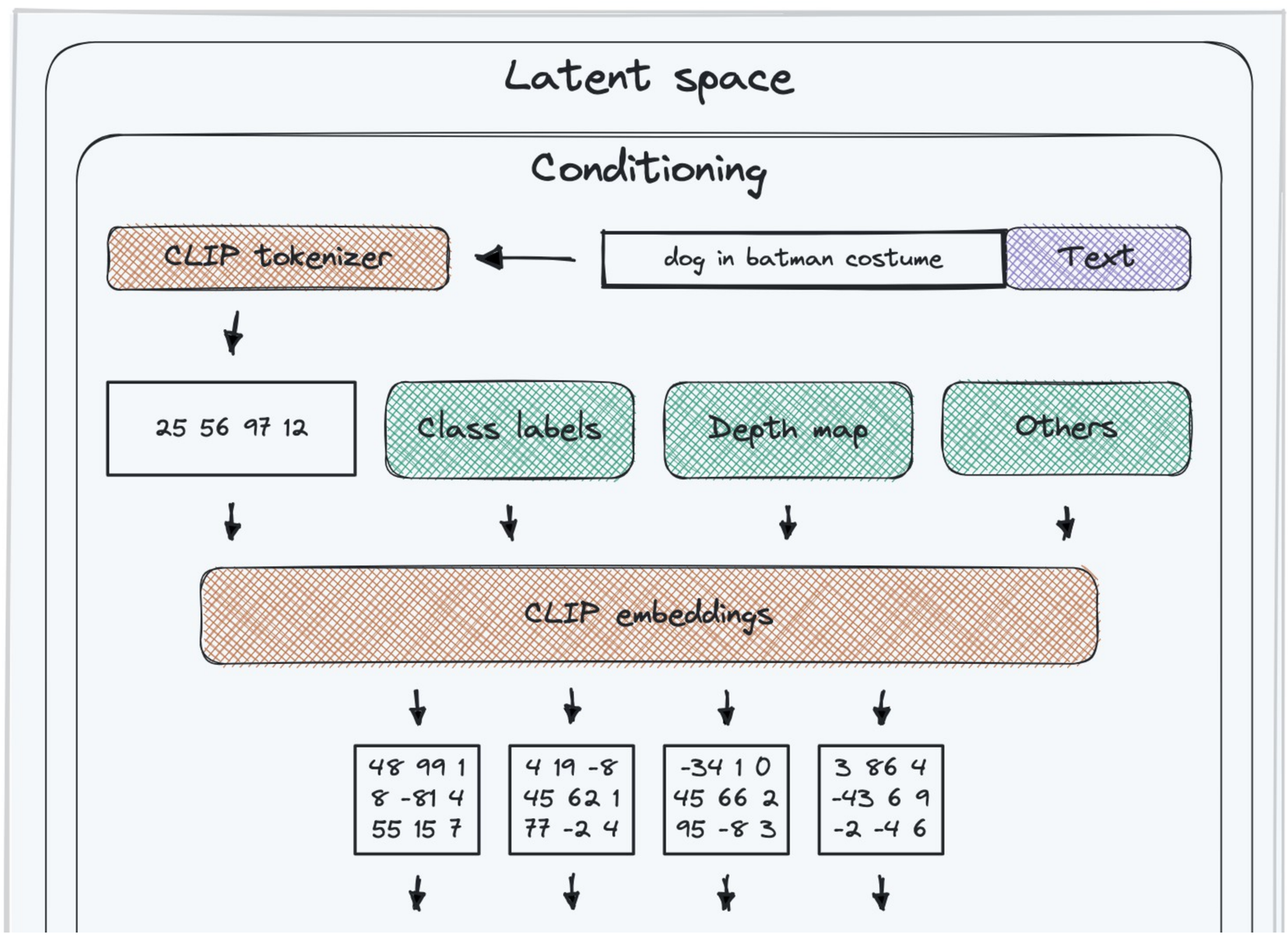
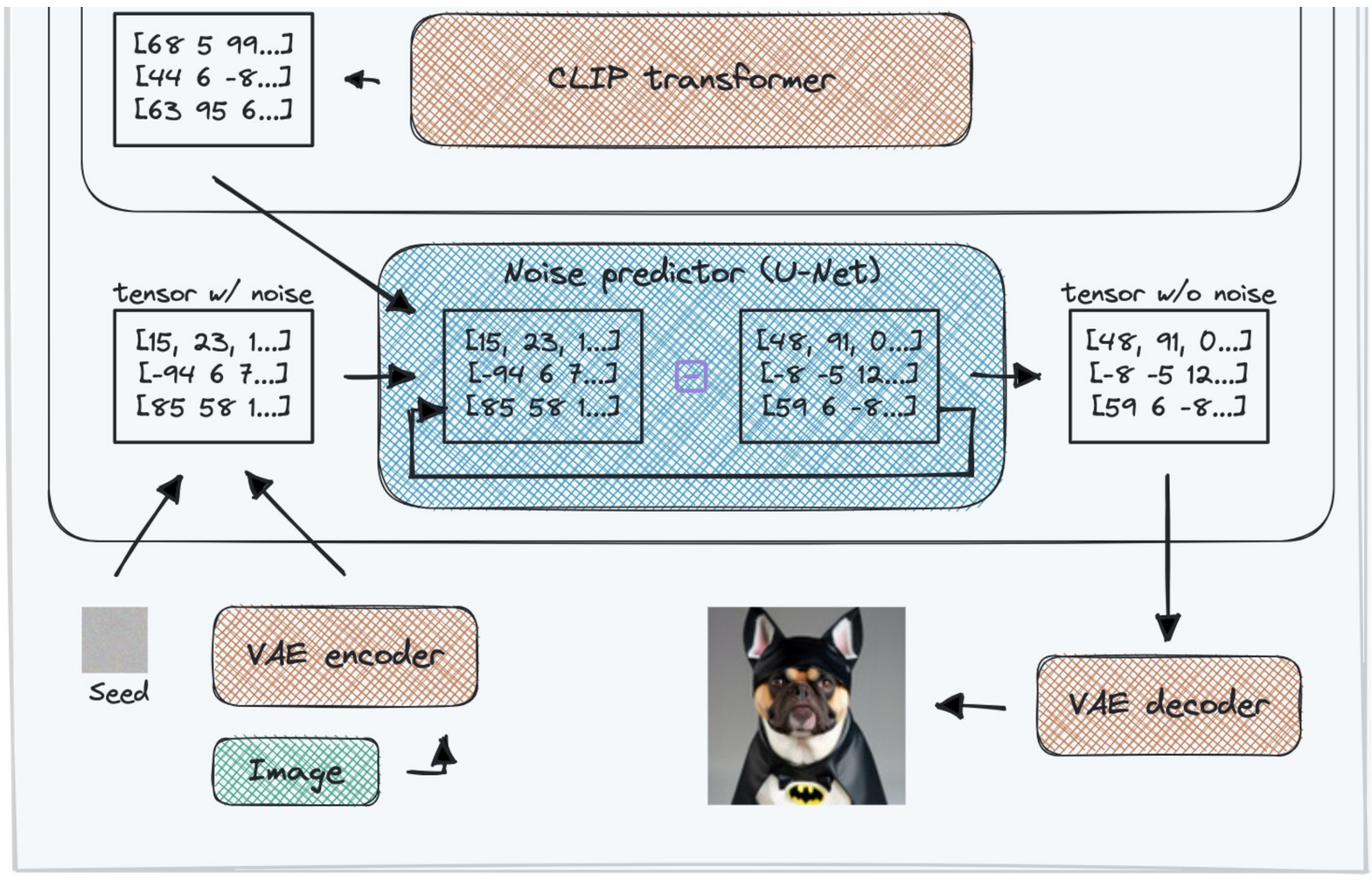


TABLE OF CONTENTS



h2 Model structure

Before starting, it is useful to become familiar with its structure. Remember that Stable Diffusion contains several components inside:

- **Tokenizer**: converts text into tokens.
- **Transformer**: transforms embeddings through attention mechanisms.
- **Variational autoencoder** (VAE): converts images into tensors within the latent space and vice versa.
- **U-Net**: predicts the noise of a tensor.
- **Scheduler**: guides the noise predictor and samples images with less noise at each step.
- Other models, such as the NSFW filter.

These models are distributed in various formats, the most common being `ckpt` and `safetensors` (since they aggregate all the components in a single file). Thanks to Hugging Face the models are also available in `diffusers` format, for easy use with their library of the same name. This format consists of **several folders with all the components separately**, perfect to understand their composition.

If we browse the Stable Diffusion 1.5 repository we will find the following structure:

You can see which scheduler, tokenizer, transformer, U-Net or VAE Stable Diffusion 1.5 uses by simply exploring the `.json` files you will find inside these folders.

Here you will find the individual models in `.bin` or `.safetensors` format. Both with the `fp16` variant which, unlike `fp32`, uses **half the disk space and memory** thanks to a decrease in decimal number precision, with hardly any effect on the final result.

Installation of libraries

First of all, make sure you have Python 3.10.

Also, if you have NVIDIA graphics and you are going to use CUDA to speed up the process (in this article I will use it), you will need to install CUDA Toolkit. You can follow these steps for its installation.

Now, we create a virtual environment and install the necessary libraries:

Create the virtual environment

```
> python -m venv .venv
```

Enable virtual environment

```
# Unix
> source .venv/bin/activate

# Windows
> .venv\Scripts\activate
```

Install required libraries

```
# If you are not going to use CUDA, remove the --index-url parameter
> pip install torch torchvision --index-url https://download.pytorch.org/whl/cu118
> pip install transformers accelerate diffusers tqdm pillow
```

Inference process

We can now create a file (for example `inference.py`), where we will write the code of our application.

Blog repository

If you want to copy and paste the entire code, remember that it is available at articles/how-to-implement-stable-diffusion/inference.py.

In the blog repository on GitHub you will find all the content associated with this and other articles.

Import what we need

The first thing is to import the libraries and methods that we are going to use:

Python

```
import torch
from torchvision.transforms import ToPILImage
from transformers import CLIPTextModel, CLIPTokenizer
from diffusers import AutoencoderKL, UNet2DConditionModel, EulerDiscreteScheduler
from tqdm.auto import tqdm
from PIL import Image
```

Later you will understand what each thing is for.

Instantiate models

We are going to instantiate the necessary models in order to have them available throughout the application.

We need the tokenizer, the transformer (text encoder), the noise predictor (U-Net), the scheduler and the variational autoencoder (VAE). All models are obtained from the Stable Diffusion 1.5 Hugging Face repository.

Each model is extracted from a specific folder (subfolder) and we make use of the `.safetensors` format when available. In addition, **the parameterized models are moved to the graphics card** using `to('cuda')`, to speed up the computations.

```python
tokenizer = CLIPTokenizer.from_pretrained(
  'runwayml/stable-diffusion-v1-5',
  subfolder='tokenizer',
)

text_encoder = CLIPTextModel.from_pretrained(
  'runwayml/stable-diffusion-v1-5',
  subfolder='text_encoder',
  use_safetensors=True,
).to('cuda')

scheduler = EulerDiscreteScheduler.from_pretrained(
  'runwayml/stable-diffusion-v1-5',
  subfolder='scheduler',
)

unet = UNet2DConditionModel.from_pretrained(
  'runwayml/stable-diffusion-v1-5',
  subfolder='unet',
  use_safetensors=True,
).to('cuda')

vae = AutoencoderKL.from_pretrained(
  'runwayml/stable-diffusion-v1-5',
  subfolder='vae',
  use_safetensors=True,
).to('cuda')
```

Stable Diffusion 1.5 uses the scheduler `PLMS` (also called `PNDM`), but we are going to use `Euler` to show how easy it is to replace it (we have already done so).

Initialize parameters

Next, we define the parameters we need for image generation.

Let's define our prompt. We use a list in case we would like to generate several images at the same time using different prompts (`['prompt1', 'prompt2', '...']`) or several images of the same prompt using a random seed (`['prompt'] * 4`). For now we are not going to complicate things, we use a single prompt:

```python
prompts = ['silly dog wearing a batman costume, funny, realistic, canon, award w
```

To make the code more readable, we store how many images we are going to generate at the same time:

```python
batch_size = 1
```

For the sampling process, we specify that we are going to use 30 steps (steps or sampling steps). That is, the image will be denoised 30 times.

```python
inference_steps = 30
```

To obtain a reproducible result we specify a seed instead of being random. This number will be used later to generate a tensor with noise from which we will clean the image until we obtain the result. **If we start from the same noise, we will always get the same result**.

```python
seed = 1055747
```

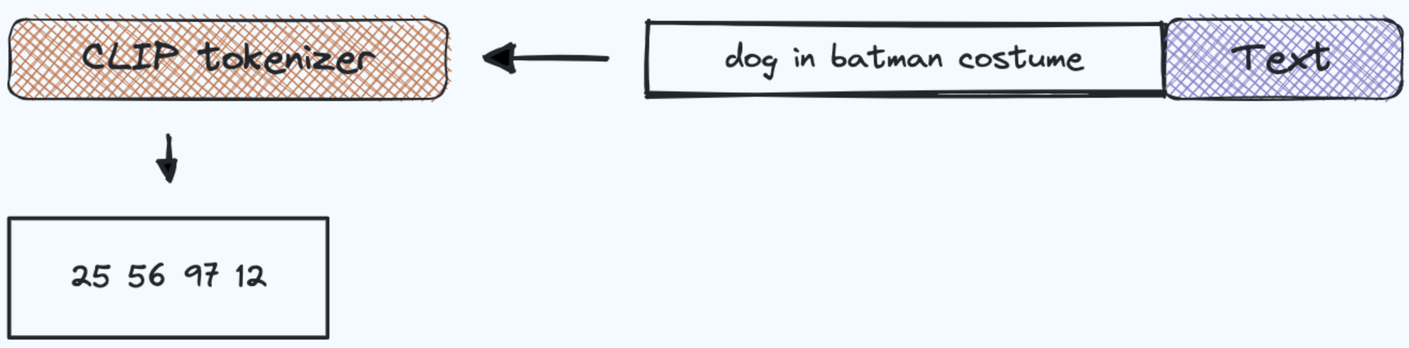
And finally, we also save in some variables the value of CFG, as well as the size of the image we want to generate:

```python
cfg_scale = 7
height = 512
width = 512
```

Conditioning

Let's start by generating the tensor that contains the information to guide the noise predictor towards the image we expect to obtain.

Tokenizer

> Since computers do not understand letters, the first task is to use a **tokenizer** to convert each word into a number called a **symbol** (**token**).
>
> 
>
>
> How Stable Diffusion works

Let's convert a test prompt into tokens:

```python
print(tokenizer('dog in batman costume'))
```

```
{'input_ids': [49406, 1929, 530, 7223, 7235, 49407], 'attention_mask': [1, 1, 1, 1, 1, 1]
```

It has returned a dictionary where `input_ids` is a list with the following tokens: `49406, 1929, 530, 7223, 7235, 49407` .

If you open the file vocab.json that you will find in the `tokenizer` folder, you will find a **dictionary** that assigns tokens to all possible terms (remember, they don't always have to be words).

So, we can see how this prompt has been tokenized:

vocab.json


```
"◁startoftext▷": 49406,
"dog</w>": 1929,
"in</w>": 530,
"batman</w>": 7223,
"costume</w>": 7235,
"◁endoftext▷": 49407,
```


Easy, isn't it?

> “
> [...] tokens are stored in a vector that has a size of 77 tokens (1x77).
>
> How Stable Diffusion works

As we saw in the previous article, **the vector has to have a size of 77 tokens**. If this limit is exceeded, they can be eliminated or solved with concatenation and feedback techniques to use all tokens. In this example we only have 6 and in our real prompt we don't have 77 either. Where do we get the rest? Let's welcome **padding** and **truncation**.

Padding is the technique that inserts a special token to **fill in the missing elements**. Truncation, on the other hand, is a technique that **remove tokens** when the desired amount is exceeded.

So let's tokenize our prompt as follows:

Python

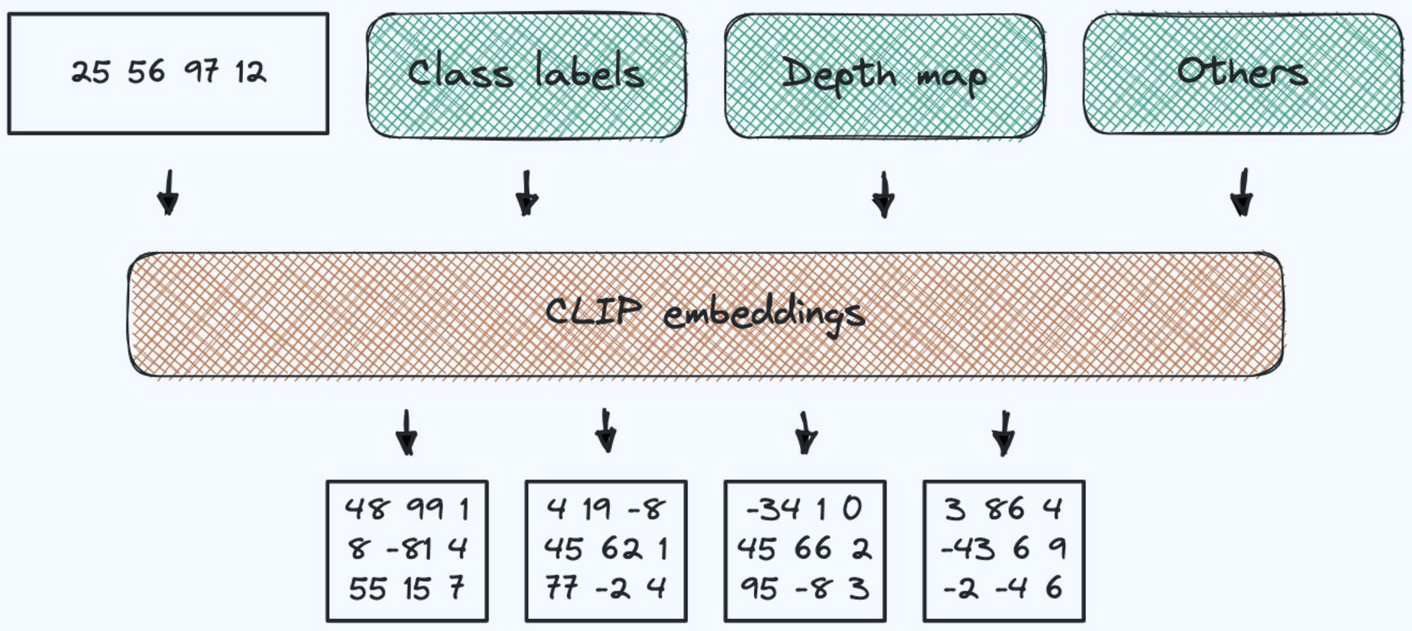
```python
cond_input = tokenizer(
  prompts,
  max_length=tokenizer.model_max_length,   # Size we need (77)
  padding='max_length',                    # Apply padding if necessary
  truncation=True,                         # Apply truncation if necessary
  return_tensors='pt',
)

print(cond_input)
```

```
{'input_ids': tensor([[49406,  1929,   530,  7223,  7235, 49407, 49407, 49407, 49407, 494
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407]]), 'attention_mask': tensor([[1
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0]])}
```

Now we can see how `input_ids` contains 77 elements. The token `49407` has been repeated as many times as necessary. Additionally, `input_ids` is now a tensor thanks to the `return_tensors='pt'` argument.

Embedding

> It is worth noting that each token will contain 768 dimensions. That is, if we use the word `car` in our prompt, that token will be converted into a 768-dimensional vector. Once this is done with all the tokens we will have an embedding of size 1x77x768.



How Stable Diffusion works

Simple task for us. We call the `text_encoder()` function, passing it the `input_ids` property of the tensor as an argument and keeping the first element it returns.

```python
with torch.no_grad():
  cond_embeddings = text_encoder(cond_input.input_ids.to('cuda'))[0]

print(cond_embeddings)
```

```
tensor([[[-0.3884,  0.0229, -0.0522,  ..., -0.4899, -0.3066,  0.0675],
         [-1.8022,  0.5477,  1.0725,  ..., -1.5483, -0.5022, -0.2065],
         [-0.3402,  1.4715, -0.7753,  ..., -1.0974, -0.6557,  0.0747],
         ...,
         [-0.9392,  0.1777,  0.2575,  ...,  0.9130, -0.3660, -1.0388],
         [-0.9334,  0.1780,  0.2590,  ...,  0.9113, -0.3683, -1.0343],
         [-0.8973,  0.1848,  0.2609,  ...,  0.9189, -0.3297, -1.0798]]],
       device='cuda:0')
```

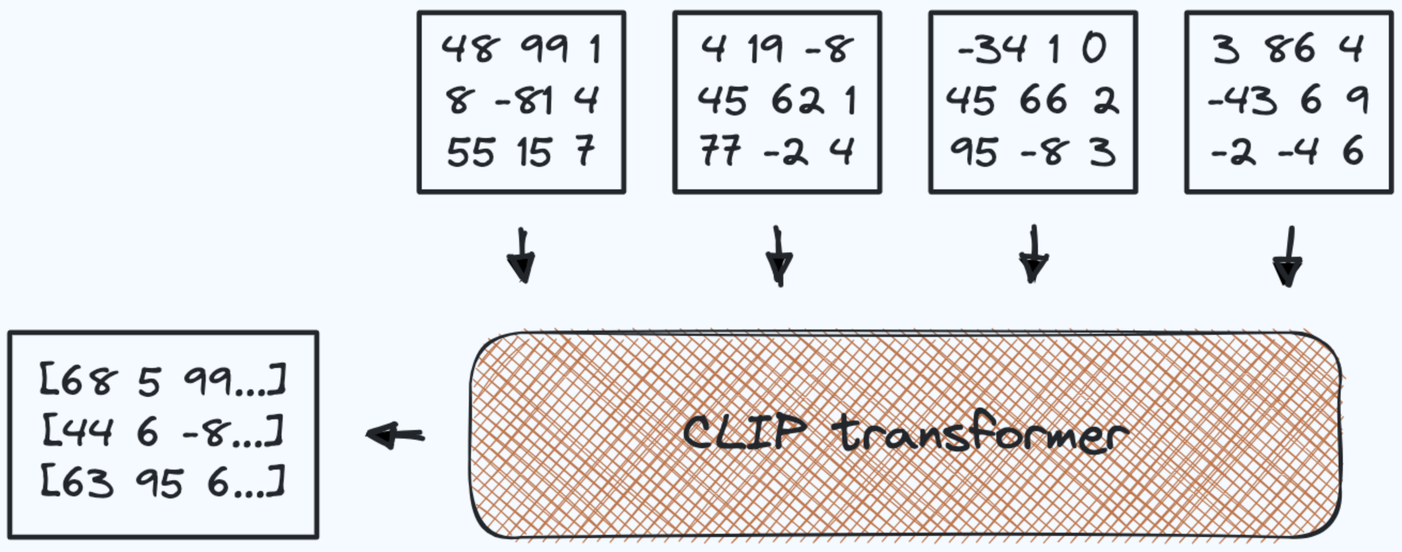
Since we are using CUDA we have to send the tensor stored in `cond_input.input_ids` to the graphics card using `to('cuda')`.

The line `with torch.no_grad()` disables the automatic gradient calculation. Without going into detail, it is something we don't need for the inference process and we **avoid using memory unnecessarily**. We will enter this context every time we make use of a parameterized model.

We now have our embedding ready.

Transformer

> This is the last step of the conditioning. In this part **the embeddings are processed** by a CLIP transformer model.



How Stable Diffusion works

Don't feel cheated, but CLIP's `text_encoder()` already takes care of applying the **attention mechanisms** when creating the embedding, so you don't have to do anything else.

With this we finish the conditioning.

Unconditioning

Stable Diffusion must also be provided with an **unconditioned embedding**. It is done in the same way but the prompt is an empty string as many times as there are images we are generating at the same time.

```python
uncond_input = tokenizer(
  [''] * batch_size,
  max_length=tokenizer.model_max_length,
  padding='max_length',
  truncation=True,
  return_tensors='pt',
)

with torch.no_grad():
  uncond_embeddings = text_encoder(uncond_input.input_ids.to('cuda'))[0]
```

Python

We join these embeddings, both conditioned and unconditioned, into a single tensor:

```python
text_embeddings = torch.cat([uncond_embeddings, cond_embeddings])
```

Python

Negative Prompt

Do you want to add a negative prompt? This is the unconditional embedding!

When we use a positive prompt we are **guiding the noise predictor in that direction**. If we tell it that we want a `bouquet of roses`, the noise predictor will go in that direction.

The unconditioned prompt **directs the noise predictor** away from those tokens. If we didn't use it, the quality would be severely affected because it wouldn't know where to move away from.

By using an empty unconditioned prompt we are giving it extra noise. It's like telling it **to move away from the noise**. And what's the opposite? A quality image.

If instead of noise we use the unconditioned embedding to add words (negative prompt), we will be even more specific in moving the noise predictor away from the noise. If we use the negative prompt `red, pink`, what we are telling it is **to move away from colors red and pink**, so it will most likely generate an image with a bouquet of white, blue or yellow roses.

In this article we are not going to use negative prompt but I recommend that you always add one to obtain much better quality in the result. If you use a prompt like `bad quality, deformed, oversaturated`, you will be distancing it from all that and the model will look for the opposite.

Generate a tensor with noise

> At the beginning of the process, instead of generating a noise-filled image, **latent noise** is generated and stored in a tensor.
>
> How Stable Diffusion works

To generate noise we instantiate a generator using torch.Generator and assign it the seed from which we will start:

```python
generator = torch.Generator(device='cuda')
generator.manual_seed(seed)
```

Next, we use the function torch.randn to obtain a tensor with noise. The parameter it receives is a **sequence of integers that defines the shape of the tensor**:

```python
latents = torch.randn(
  (batch_size, unet.config.in_channels, height // 8, width // 8),
  generator=generator,
  device='cuda',
)
```

We are passing `(1, 4, 64, 64)` as value. These numbers come from:

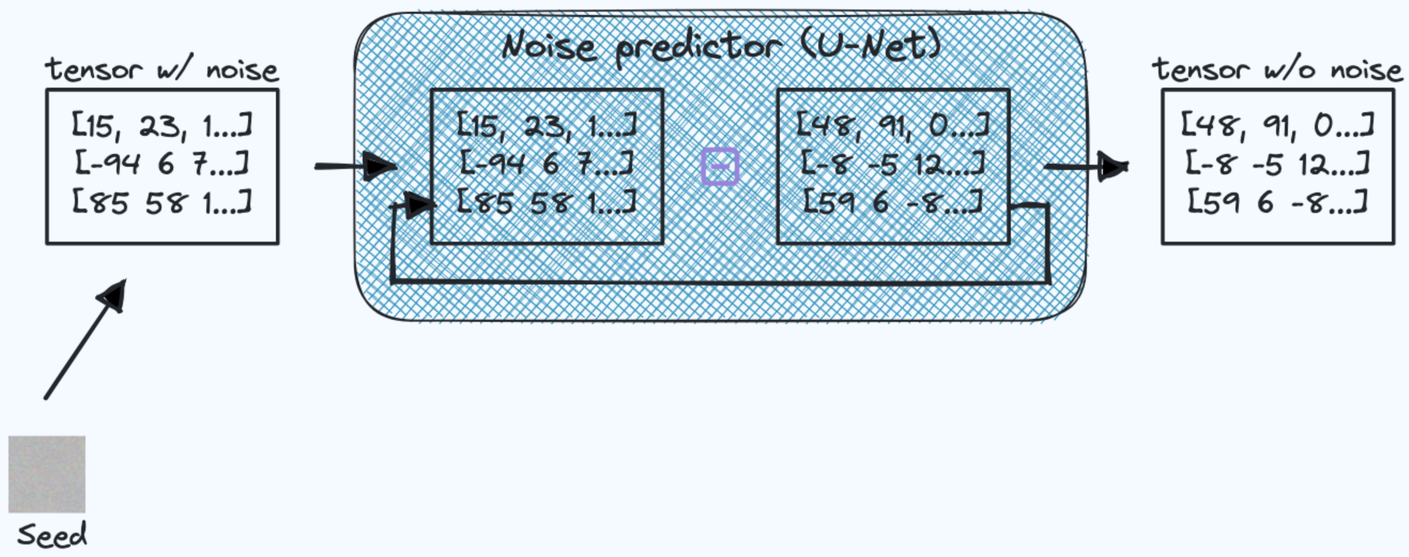
- `1`: The value of `batch_size`. That is, how many images we generate at once.
- `4`: The number of input channels that the noise predictor neural network (U-Net) has.
- `64` / `64`: The size of the image in latent space (height / width). Our image is `512×512` in size but in latent space it occupies 8 times less. This divider is specified in the Stable Diffusion 1.5 architecture.

Remember that when using CUDA, we have specified it in both functions using the `device='cuda'` argument.

Now `latents` is a tensor with noise on which we can now work **as if it were a canvas**.

h3 Clean up the tensor noise

> The noise predictor estimates how much noise is in the image. After this, the algorithm called **sampler** generates an image with that amount of noise and it is subtracted from the original image. This process is repeated the number of times specified by **steps** or **sampling steps**.



How Stable Diffusion works

Let's configure the scheduler to specify in how many steps we want to clean the tensor:

```python
scheduler.set_timesteps(inference_steps)
```

We can check how it works internally by printing the following property:

```python
print(scheduler.timesteps)
```

```
tensor([999.0000, 964.5517, 930.1035, 895.6552, 861.2069, 826.7586, 792.3104,
        757.8621, 723.4138, 688.9655, 654.5172, 620.0690, 585.6207, 551.1724,
        516.7241, 482.2758, 447.8276, 413.3793, 378.9310, 344.4828, 310.0345,
        275.5862, 241.1379, 206.6897, 172.2414, 137.7931, 103.3448,  68.8966,
         34.4483,   0.0000])
```

As there are 30 steps, **30 elements** have been generated separated by the same distance (`34.4483` units).

Some schedulers, such as `DPM2 Karras` or `Euler` , require that from the first step the tensor values are already multiplied by the standard deviation of the initial noise distribution. Schedulers that do not need it will simply multiply by `1` . That is... we need to add the following operation:

```python
latents = latents * scheduler.init_noise_sigma
```

We already have the tensor with noise, the conditioning and the scheduler. With this we can now generate the loop that will clean the tensor over 30 turns. We use the tqdm library to display a progress bar. I will explain each line directly in the code.

```python
for t in tqdm(scheduler.timesteps):
  # Since we are using classifier-free guidance, we duplicate the tensor to avoid making
  # One pass will be for the conditioned values and another for the unconditioned ones
  # With this it gains efficiency
  latent_model_input = torch.cat([latents] * 2)

  # This line ensures compatibility between various schedulers
  # Basically, the ones that need to scale the input based on the timestep
  latent_model_input = scheduler.scale_model_input(latent_model_input, timestep=t)

  with torch.no_grad():
    # The U-Net is asked to make a prediction of the amount of noise in the tensor
    noise_pred = unet(latent_model_input, t, encoder_hidden_states=text_embeddings).sampl

  # We assign half of the estimated noise to conditioning and the other half to unconditi
  noise_pred_uncond, noise_pred_cond = noise_pred.chunk(2)

  # Here the prediction is adjusted to guide towards a conditioned result
  # That is, more or less importance is given to the conditioning (to the prompt in this
  noise_pred = noise_pred_uncond + cfg_scale * (noise_pred_cond - noise_pred_uncond)

  # A new tensor is generated by subtracting the amount of noise we have previously calcu
  # This is the process that cleans the noise until all the scheduler steps have finished
  latents = scheduler.step(noise_pred, t, latents).prev_sample
```

Python

After finishing this loop, we have our tensor free of noise and ready to be converted into a spectacular image. Or maybe in a churro, now we will find out.

Reproducibility

If even though you are using the same seed you are not getting the same image as a result, this **problem of reproducibility** is probably because you are using an ancestral or stochastic sampler.

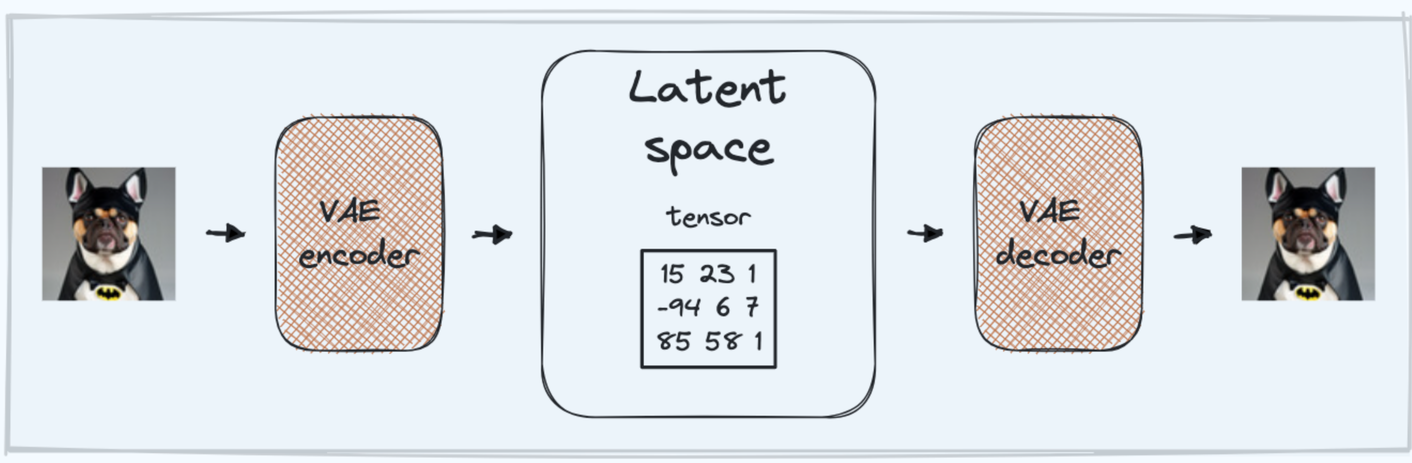
This happens because ancestral samplers add extra noise at each step and stochastic samplers use information from the previous step. Therefore, they need to have access to the generator to provide this variability at each step. You just have to add it to this line:

```python
latents = scheduler.step(noise_pred, t, latents, generator=generator).pr
```

Python l

h3 Converting the tensor into an image

> A **variational autoencoder** (**VAE**) is a type of neural network that converts an image into a tensor in latent space (**encoder**) or a tensor in latent space into an image (**decoder**).



How Stable Diffusion works

There's little left. The scale factor (`vae.config.scale_factor`) of the VAE itself must be taken into account, a value set at `0.18215` . Once the tensor is normalized we can decode it using `vae.decode()` to **take it out of latent space and put it into image space**.

```python
latents = latents / vae.config.scale_factor
with torch.no_grad():
  images = vae.decode(latents).sample
```

Python

We still have a tensor, but it is no longer inside the Matrix. This tensor has values ranging from `-1` to `1` , so we first normalize it to a range from `0` to `1` .

We could perform calculations to convert these values into RGB values but let's allow torchvision to handle that. We use its ToPILImage transformation to convert the `torch` tensor into a `pillow` image (or multiple images, depending on the `batch_size`). The `save()` method from Pillow will take care of saving the images to disk.

```python
images = (images / 2 + 0.5).clamp(0, 1)

to_pil = ToPILImage()

for i in range(1, batch_size + 1):
  image = to_pil(images[i])
  image.save(f'image_{i}.png')
```

Python

Run `python inference.py` ... and we can now see our **gorgeous and perfect image**!

Well... okay, it's a churro. It would be necessary to improve the prompt, change the model...

Conclusion

We have seen what components a Stable Diffusion model is composed of and also what results they produce so that we can connect them together. Thanks to the `diffusers` library we have been able to abstract the code enough to not have to reinvent the wheel from scratch, but also not remain on the surface without understanding anything.

In `diffusers` we have pipelines like the one in Stable Diffusion to run the inference process in a couple of lines:

```python
import torch
from diffusers import StableDiffusionPipeline

pipe = StableDiffusionPipeline.from_pretrained('runwayml/stable-diffusion-v1-5').to('cuda
image = pipe('dog in batman costume').images[0]
```

It could be said that we have implemented our own pipeline, in which we can interchange components such as the scheduler or the VAE.

I hope you didn't find it too complicated and that this article has been helpful to understand how the Stable Diffusion inference process works.