Student Write-up (Images rendered above)

NOTE : Images are rendered in the above cells !

The best use of hyperparameters:

1. guidance scale = 7.5
2. num_steps = 50
3. scheduler = DPMSolverMultistepScheduler++
4. seed = [42,70]

With respect to guidance scale, I observed that too less of a guidance scale leads to an image away from the textual prompt, and too much of it causes it to be rendered in an abstract cartoonish style.

For num_steps, as I increase the number of time steps, the model converges well but the change is mostly pronounced between 10 to 50 as compared to 50 to 100 as noted from the above images

With respect to scheduler, the best one with fastest average run time was DPMSolver Multi-step, whereas PNDM gave visually non-appealing results.

Written Questions

1. At the end of the derivation of ELBO from lecture, we are left with two parts:

$$\log p(x) = \mathbb{E}_q[\log \frac{p(x,z)}{q(z|x)}] + \log \mathbb{E}_q[\frac{q(z|x)}{p(z|x)}]$$

- Which of these (left or right), becomes the KL Divergence term?
- Write out the full equation (including the KL divergence term) for what we would want to maximize. Note you can use LaTeX in colab by adding $$ signs before and after the equation.
- Intuitively explain what the KL Divergence term is measuring

Answer:

1. The term towards the right is KL Divergence (ELBO being left).
2. The full equation:

$$\log p_\theta(x) = E_{q_\phi}(z \mid x)\left[\log \frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right] + D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z \mid x)\right)$$

$$\log p_\theta(x) \geq E_{q_\phi}(z \mid x)\left[\log \frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right] \longrightarrow ELBO$$

$$D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z \mid x)\right) = E_{q_\phi}(z \mid x)\left[\log\left(\frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right)\right]$$

$$\max ELBO = \log p_\theta(x) - D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z \mid x)\right)$$

In the above equations,both the maximum log likelihood and KL Divergence are intractable. And we know that the KL Divergence cannot be negative, it must be greater than or equal to zero. Upon inspecting the equation, we see that maximizing ELBO is the same as minimizing the KL divergence between the two distributions. As our goal is to minimize KL divergence, we would want to maximize ELBO.

3. Intuitively, it quantifies the difference between two probability distributions by measuring the information lost when the second distribution is used to approximate the first one. It sort of measures the average amount of information needed to specify which event occurred based on the second distribution compared to the optimal encoding based on the first distribution. $p_\theta(z \mid x)$, the posterior, is the groud truth. We are approximating the posterior with another distribution, $q_\phi(z \mid x)$.Our goal is to minimize the divergence between the approximation and the posterior so that the approximation is as close as

possible.

---

2. We've written out part of the derviation for ELBO for a heirarchical VAE/diffusion model below. Identify the mistakes in these steps, if there are any, and write the correct equation. If there are no mistakes, put "No mistakes"

$$\log p(x) = \mathbb{E}_q[\log \frac{p(x, z_{1:T})}{q(z_{1:T}|x)}]$$

$$\log p(x) = \mathbb{E}_q[\log \frac{p(z_T)p(x|z_1)\Pi_{t=2}^{T}p(z_t|z_{t-1})}{q(z_1|x)\Pi_{t=2}^{T}q(z_{t-1}|z_t)}]$$

Answer:

The above equation is wrong.

We assign the distribution $p\left(z_{t-1}, z_t\right)$ to the process of reverse diffusion (denoising), for an arbitary t and the distribution $q\left(z_t, z_{t-1}\right)$ to the process of forward diffusion. Using Markov property, each node depends on it's neighbouring node in the following way:

$$p\left(x, z_{1:T}\right) = p\left(z_T\right) p_\theta\left(x \mid z_1\right) \prod_{t=2}^{T} p_\theta\left(z_{t-1} \mid z_t\right)$$

$$q\left(z_{1:T} \mid x\right) = q_\phi\left(z_1 \mid x\right) \prod_{t=2}^{T} q_\phi\left(z_t \mid z_{t-1}\right)$$

$$\log p(x) = E_q\left[\log \frac{p\left(z_T\right) p_\theta\left(x \mid z_1\right) \prod_{t=2}^{T} p_\theta\left(z_{t-1} \mid z_t\right)}{q_\phi\left(z_1 \mid x\right) \prod_{t=2}^{T} q_\phi\left(z_t \mid z_{t-1}\right)}\right]$$

EE239 Homework 4: Diffusion Models

Stable Diffusion is a text-to-image latent diffusion model created by the researchers and engineers from CompVis, Stability AI and LAION similar to the ones we discussed in class. In this homework, we will use the HuggingFace diffusers library to build a simple Stable Diffusion pipeline and then experiment with different components. At the end, you'll provide a write-up about what worked and what didn't along with some image samples.

A lot of this is based on this notebook:
https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers/stable_diffusion.ipynb#scrollTo=gd-vX3cavOCt

```
!nvidia-smi

Sun Dec 17 05:16:26 2023
+-----------------------------------------------------------------------
------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA
Version: 12.2     |
|-----------------------------------------+----------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A |
Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage |
GPU-Util  Compute M. |
|                                         |                      |
MIG M. |
|
=========================================+======================+======
=================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   42C    P8              10W /  70W |      0MiB / 15360MiB |
0%      Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------
+----------------------+


+-----------------------------------------------------------------------
------------------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
|        ID   ID
```

```
Usage      |
|
================================================================================
=================|
|  No running processes found
|
+-------------------------------------------------------------------------------
------------------+

!pip install diffusers==0.24.0
!pip install transformers scipy ftfy accelerate

Collecting diffusers==0.24.0
  Downloading diffusers-0.24.0-py3-none-any.whl (1.8 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 9.2 MB/s eta
0:00:00
ent already satisfied: Pillow in /usr/local/lib/python3.10/dist-
packages (from diffusers==0.24.0) (9.4.0)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(3.13.1)
Requirement already satisfied: huggingface-hub>=0.19.4 in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(0.19.4)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(7.0.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(1.23.5)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(2023.6.3)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(2.31.0)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from diffusers==0.24.0)
(0.4.1)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4-
>diffusers==0.24.0) (2023.6.0)
Requirement already satisfied: tqdm>=4.42.1 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4-
>diffusers==0.24.0) (4.66.1)
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4-
>diffusers==0.24.0) (6.0.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4-
>diffusers==0.24.0) (4.5.0)
Requirement already satisfied: packaging>=20.9 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4-
>diffusers==0.24.0) (23.2)
Requirement already satisfied: zipp>=0.5 in
/usr/local/lib/python3.10/dist-packages (from importlib-metadata-
>diffusers==0.24.0) (3.17.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests-
>diffusers==0.24.0) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests-
>diffusers==0.24.0) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests-
>diffusers==0.24.0) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests-
>diffusers==0.24.0) (2023.11.17)
Installing collected packages: diffusers
Successfully installed diffusers-0.24.0
Requirement already satisfied: transformers in
/usr/local/lib/python3.10/dist-packages (4.35.2)
Requirement already satisfied: scipy in
/usr/local/lib/python3.10/dist-packages (1.11.4)
Collecting ftfy
  Downloading ftfy-6.1.3-py3-none-any.whl (53 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 53.4/53.4 kB 958.7 kB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 265.7/265.7 kB 3.6 MB/s eta
0:00:00
ent already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers) (3.13.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.19.4)
Requirement already satisfied: numpy>=1.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.0)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.1)
```

```
Requirement already satisfied: tqdm>=4.27 in
/usr/local/lib/python3.10/dist-packages (from transformers) (4.66.1)
Requirement already satisfied: wcwidth<0.3.0,>=0.2.12 in
/usr/local/lib/python3.10/dist-packages (from ftfy) (0.2.12)
Requirement already satisfied: psutil in
/usr/local/lib/python3.10/dist-packages (from accelerate) (5.9.5)
Requirement already satisfied: torch>=1.10.0 in
/usr/local/lib/python3.10/dist-packages (from accelerate)
(2.1.0+cu121)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.16.4->transformers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.16.4->transformers) (4.5.0)
Requirement already satisfied: sympy in
/usr/local/lib/python3.10/dist-packages (from torch>=1.10.0-
>accelerate) (1.12)
Requirement already satisfied: networkx in
/usr/local/lib/python3.10/dist-packages (from torch>=1.10.0-
>accelerate) (3.2.1)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.10/dist-packages (from torch>=1.10.0-
>accelerate) (3.1.2)
Requirement already satisfied: triton==2.1.0 in
/usr/local/lib/python3.10/dist-packages (from torch>=1.10.0-
>accelerate) (2.1.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2023.11.17)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.10.0-
>accelerate) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.10.0-
>accelerate) (1.3.0)
Installing collected packages: ftfy, accelerate
Successfully installed accelerate-0.25.0 ftfy-6.1.3
```

Stable Diffusion Pipeline

`StableDiffusionPipeline` is an end-to-end inference pipeline that you can use to generate images from text.

First, we load the pre-trained weights of all components of the model. In this notebook we use Stable Diffusion version 1.4 (CompVis/stable-diffusion-v1-4).

We load the weights from the half-precision branch `fp16` and also tell `diffusers` to expect the weights in float16 precision by passing `torch_dtype=torch.float16`.

```
import torch
from diffusers import StableDiffusionPipeline

pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-
diffusion-v1-4", torch_dtype=torch.float16)
```

```
The cache for model files in Transformers v4.22.0 has been updated.
Migrating your old cache. This is a one-time only operation. You can
interrupt this and resume the migration later on by calling
`transformers.utils.move_cache()`.
```

{"model_id":"0b6269680f944eb5895369275dc67324","version_major":2,"vers
ion_minor":0}

{"model_id":"acf6cd8d2e404072afcc2d3e628c2f9d","version_major":2,"vers
ion_minor":0}

{"model_id":"f41ec2e12e5b4c88916d1a6e66b480a8","version_major":2,"vers
ion_minor":0}

{"model_id":"af933c683eaf4043ac24ec8831dae27e","version_major":2,"vers
ion_minor":0}

{"model_id":"63a248139d9b487f97f99f4610b27ddf","version_major":2,"vers
ion_minor":0}

{"model_id":"410feec8b2864576941d4a918565a889","version_major":2,"vers
ion_minor":0}

{"model_id":"b0723d3407a2476db83597a9981862db","version_major":2,"vers
ion_minor":0}

{"model_id":"e8d9e4f0f5f84b47b741cc70d721c3b1","version_major":2,"vers
ion_minor":0}

{"model_id":"42455543df044659bb7243996fe41cca","version_major":2,"vers
ion_minor":0}

{"model_id":"d9e7c5db43c34599b23b927ae4068a4c","version_major":2,"vers
ion_minor":0}

```
{"model_id":"998f5fead30d4282b949dc1a28047744","version_major":2,"version_minor":0}

{"model_id":"14937ba1b22640bb83966439db74a178","version_major":2,"version_minor":0}

{"model_id":"f9e86310cc4d4093afbf3e4747b4d78d","version_major":2,"version_minor":0}

{"model_id":"976499b6424b4ddbae7442deedd7e3a3","version_major":2,"version_minor":0}

{"model_id":"1838d800a71d44fb819fb7b1c5aaeaa4","version_major":2,"version_minor":0}

{"model_id":"170521ce9ebd491f824cb39794a208a6","version_major":2,"version_minor":0}

{"model_id":"5e2906b18cdc49348a6488b15042e612","version_major":2,"version_minor":0}

{"model_id":"8a10820be9c84903a89ceeda002309ec","version_major":2,"version_minor":0}

{"model_id":"94e6638baedd4cd1ab0f0ea7333f8d75","version_major":2,"version_minor":0}

`text_config_dict` is provided which will be used to initialize
`CLIPTextConfig`. The value `text_config["id2label"]` will be
overriden.
`text_config_dict` is provided which will be used to initialize
`CLIPTextConfig`. The value `text_config["bos_token_id"]` will be
overriden.
`text_config_dict` is provided which will be used to initialize
`CLIPTextConfig`. The value `text_config["eos_token_id"]` will be
overriden.
```

Next, let's move the pipeline to GPU to have faster inference.

```
pipe = pipe.to("cuda")
```

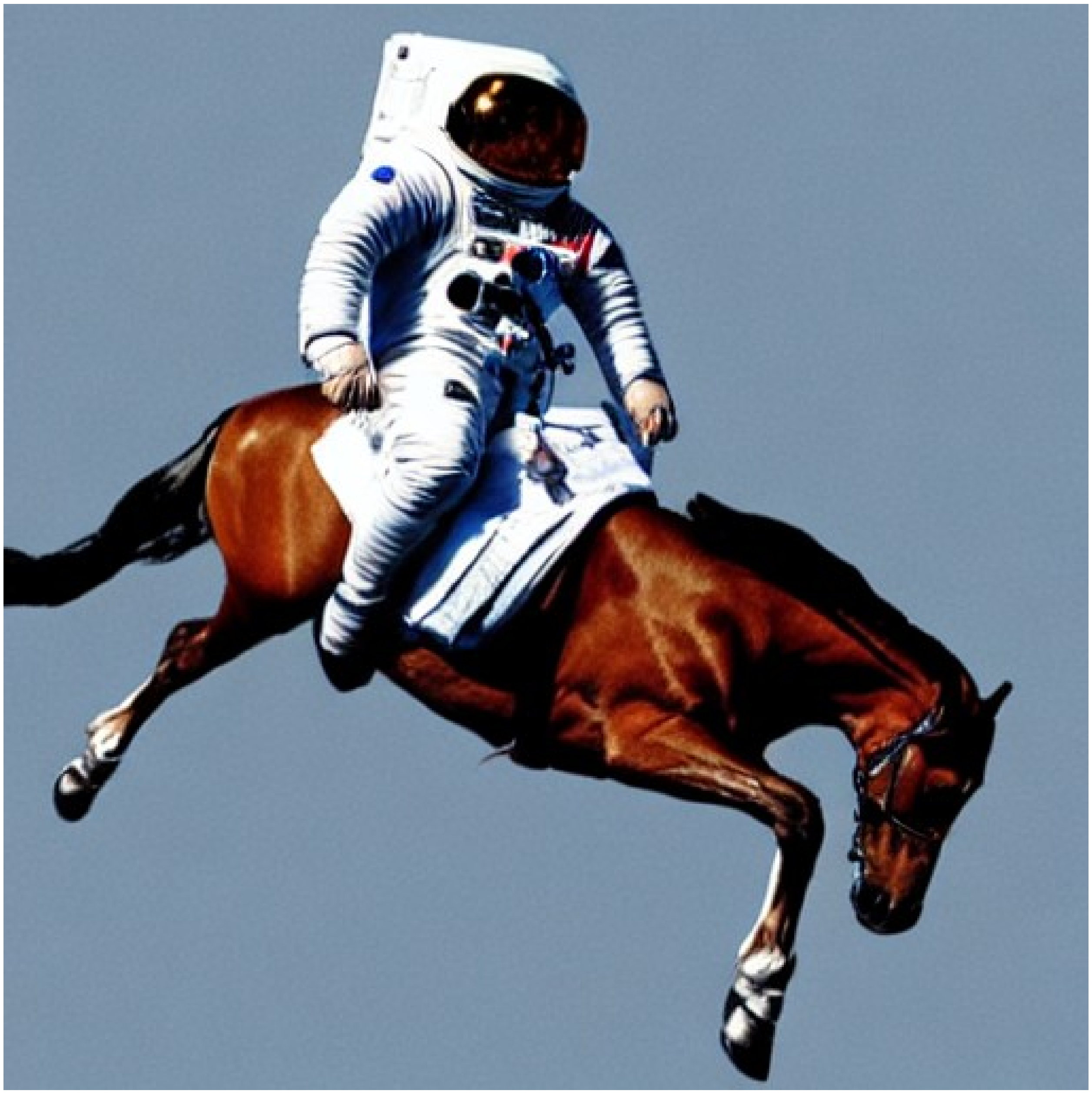
And we are ready to generate images:

```
prompt = "a photograph of an astronaut riding a horse"
image = pipe(prompt).images[0]  # image here is in [PIL format]
(https://pillow.readthedocs.io/en/stable/)

image

{"model_id":"45f299e9c24e404bb9c1c5f709b630a6","version_major":2,"version_minor":0}
```

Running the above cell multiple times will give you a different image every time. If you want deterministic output you can pass a random seed to the pipeline. Every time you use the same seed you'll have the same image result.

```
import torch

generator = torch.Generator("cuda").manual_seed(1024)

image = pipe(prompt, generator=generator).images[0]

image
```

```
{"model_id":"a435cd2de71f42f3aef7216a46790bbb","version_major":2,"vers
ion_minor":0}
```



We can change the number of denoising steps using the `num_inference_steps` parameter. Generally, increasing the step count will generate higher quality images but will take longer while lower step counts could generate worse looking or more incoherent images.

```python
h = pipe(prompt, num_inference_steps=15, generator=generator).images
```

```
{"model_id":"1a1d680935cb44bf8375783fd72194b8","version_major":2,"vers
ion_minor":0}
```

```python
import torch

generator = torch.Generator("cuda").manual_seed(1024)

image = pipe(prompt, num_inference_steps=15,
generator=generator).images[0]

image
```

{"model_id":"347bac12b2d94ec2a982fa672d07adf8","version_major":2,"version_minor":0}

The other parameter in the pipeline call is `guidance_scale`. It is a way to increase the adherence to the conditional signal which in this case is text as well as overall sample quality. In simple terms classifier free guidance forces the generation to better match with the prompt. Numbers like `7` or `8.5` give good results, if you use a very large number the images might look good, but will be less diverse.

To generate multiple images for the same prompt, we simply use a list with the same prompt repeated several times. We'll send the list to the pipeline instead of the string we used before.

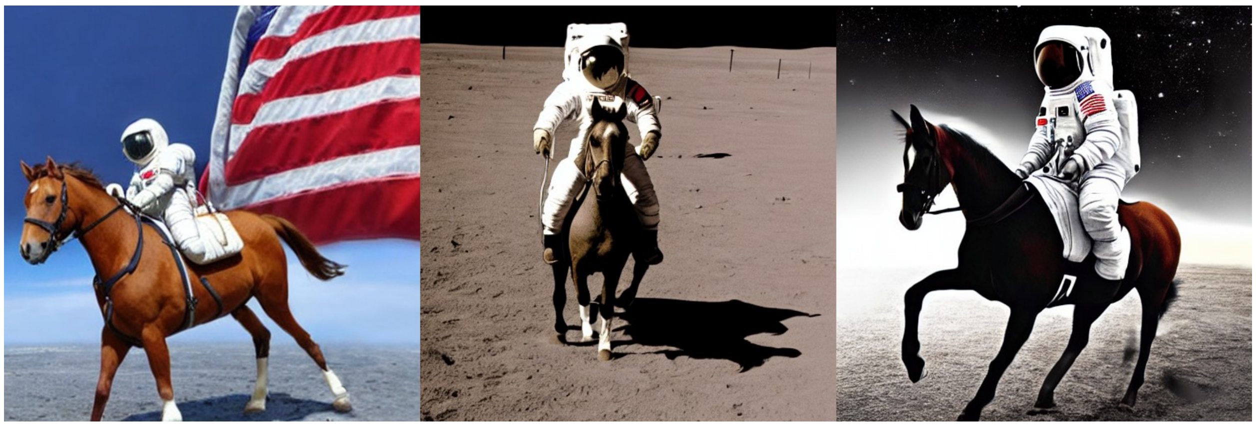
Let's first write a helper function to display a grid of images. Just run the following cell to create the `image_grid` function, or disclose the code if you are interested in how it's done.

```python
from PIL import Image

def image_grid(imgs, rows, cols):
    assert len(imgs) == rows*cols

    w, h = imgs[0][0].size
    grid = Image.new('RGB', size=(cols*w, rows*h))
    grid_w, grid_h = grid.size

    for i, img in enumerate(imgs):
        grid.paste(imgs[i][0], box=(i%cols*w, i//cols*h))
    return grid
```

Now, we can generate a grid image once having run the pipeline with a list of 3 prompts.

```python
num_images = 3
prompt = ["a photograph of an astronaut riding a horse"] * num_images

images = pipe(prompt).images

grid = image_grid(images, rows=1, cols=3)
grid
```

```
{"model_id":"b720e7c5bfae4bdca7aea8fe82ea9b91","version_major":2,"vers
ion_minor":0}
```



And here's how to generate a grid of `n × m` images.

Generate non-square images

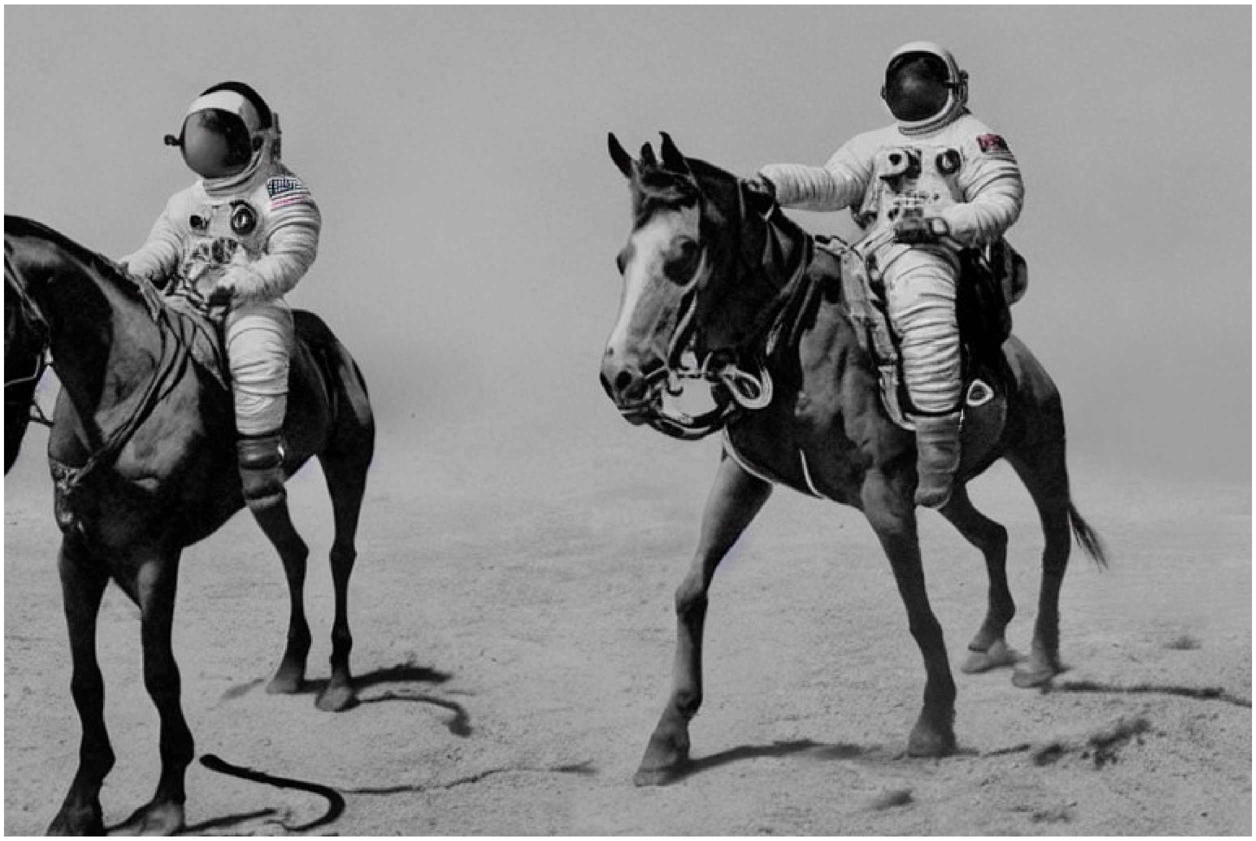
Stable Diffusion produces images of `512 × 512` pixels by default. But it's very easy to override the default using the `height` and `width` arguments, so you can create rectangular images in portrait or landscape ratios.

These are some recommendations to choose good image sizes:

- Make sure `height` and `width` are both multiples of `8`.
- Going below 512 might result in lower quality images.
- Going over 512 in both directions will repeat image areas (global coherence is lost).
- The best way to create non-square images is to use `512` in one dimension, and a value larger than that in the other one.

```
prompt = "a photograph of an astronaut riding a horse"

image = pipe(prompt, height=512, width=768).images[0]
image
```

{"model_id":"5ed11e41fea640eb8b46318f1b1197ea","version_major":2,"version_minor":0}



2. What is Stable Diffusion

In class, we covered traditional diffusion models that noise and then de-noise an image. Stable Diffusion is based on a particular type of diffusion model called **Latent Diffusion**, proposed in High-Resolution Image Synthesis with Latent Diffusion Models.

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).
2. A U-Net.
3. A text-encoder, *e.g.* CLIP's Text Encoder.

1. The 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. The 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. The Text-encoder

The text-encoder is responsible for transforming the input prompt, *e.g.* "An astronout 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.

Why is latent diffusion fast and efficient?

Since the U-Net of latent diffusion models 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 `(3, 64, 64)` in latent space, which requires `8 × 8 = 64` times less memory.

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

Stable Diffusion during inference

Putting it all together, let's now take a closer look at how the model works in inference by illustrating the logical flow.

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 64×64 where as the text prompt is transformed to text embeddings of size 77×768 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.

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.

3. Creating a pipeline with `diffusers`

Let's now build our own pipeline with components from `diffusers`. We'll use this setup to test our different components and then see how they work together.

```
import torch
torch_device = "cuda" if torch.cuda.is_available() else "cpu"
```

The pre-trained model includes all the components required to setup a complete diffusion pipeline. They are stored in the following folders:

- `text_encoder`: Stable Diffusion uses CLIP, but other diffusion models may use other encoders such as `BERT`.
- `tokenizer`. It must match the one used by the `text_encoder` model.
- `scheduler`: The scheduling algorithm used to progressively add noise to the image during training.
- `unet`: The model used to generate the latent representation of the input.
- `vae`: Autoencoder module that we'll use to decode latent representations into real images.

We can load the components by referring to the folder they were saved, using the `subfolder` argument to `from_pretrained`. We have loaded a set of base components here:

```
from transformers import CLIPTextModel, CLIPTokenizer
from diffusers import AutoencoderKL, UNet2DConditionModel,
PNDMScheduler, LMSDiscreteScheduler

# 1. Load the autoencoder model which will be used to decode the
latents into image space.
vae = AutoencoderKL.from_pretrained("CompVis/stable-diffusion-v1-4",
subfolder="vae")

# 2. Load the tokenizer and text encoder to tokenize and encode the
text.
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-
patch14")
text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-
patch14")

# 3. The UNet model for generating the latents.
unet = UNet2DConditionModel.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="unet")
scheduler = PNDMScheduler.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="scheduler")
```

```
{"model_id":"b298818293004d3f94faea12f8d43f38","version_major":2,"vers
ion_minor":0}

{"model_id":"9f36b6a33b7f48e2b1263cf8a8da955b","version_major":2,"vers
ion_minor":0}

{"model_id":"28396870b0e7486b8b361bf810d10fdc","version_major":2,"vers
ion_minor":0}

{"model_id":"7e17fb4d5c0f4275bb9828a84a0f5122","version_major":2,"vers
ion_minor":0}

{"model_id":"4be54d8fcbf843509fb671dd603bd72e","version_major":2,"vers
ion_minor":0}

{"model_id":"29e8b0e91afc4d52adc951d97f9caa43","version_major":2,"vers
ion_minor":0}

{"model_id":"581904256d294bb9853e41a6f44dca37","version_major":2,"vers
ion_minor":0}

{"model_id":"e8e4d61de05c42deab08abf95b6615c8","version_major":2,"vers
ion_minor":0}

{"model_id":"bee7beef8b26419a9ad3448a7ba66671","version_major":2,"vers
ion_minor":0}

{"model_id":"48aa260d1e654cfd918a2b3d20dbb9c9","version_major":2,"vers
ion_minor":0}
```

```
{"model_id":"a56d31dc91ce4741a237b6c638602355","version_major":2,"vers
ion_minor":0}
```

Lets now create a function that takes in all these individual components and put them together to create a full pipeline: This code is written for you.

```
from tqdm.auto import tqdm
from torch import autocast

def render_image(prompt, f_vae, f_unet, f_tokenizer, f_text_encoder,
f_scheduler, height=512, width=512,\
                 guidance_scale=7.5, num_steps=50, seed=42):
    f_vae = f_vae.to(torch_device)
    f_text_encoder = f_text_encoder.to(torch_device)
    f_unet = f_unet.to(torch_device)

    generator = torch.manual_seed(seed)

    batch_size = 1

    # Get prompt embeddings
    text_input = f_tokenizer(prompt, padding="max_length",
max_length=f_tokenizer.model_max_length, truncation=True,
return_tensors="pt")

    with torch.no_grad():
          text_embeddings =
f_text_encoder(text_input.input_ids.to(torch_device))[0]

    # Unconditional embedding
    max_length = text_input.input_ids.shape[-1]
    uncond_input = f_tokenizer([""] * batch_size,
padding="max_length", max_length=max_length, return_tensors="pt")
    with torch.no_grad():
        uncond_embeddings =
f_text_encoder(uncond_input.input_ids.to(torch_device))[0]
    text_embeddings = torch.cat([uncond_embeddings, text_embeddings])
# Combine conditional and unconditional for full embedding

    # Generate random starting latent
    latents = torch.randn((batch_size, f_unet.in_channels, height //
8, width // 8), generator=generator)
    latents = latents.to(torch_device)

    latents = latents * f_scheduler.init_noise_sigma

    f_scheduler.set_timesteps(num_steps)

    for t in tqdm(f_scheduler.timesteps):
        # expand the latents if we are doing classifier-free guidance
```

```
to avoid doing two forward passes.
        latent_model_input = torch.cat([latents] * 2)

        latent_model_input =
f_scheduler.scale_model_input(latent_model_input, t)

        # predict the noise residual
        with torch.no_grad():
            noise_pred = f_unet(latent_model_input, t,
encoder_hidden_states=text_embeddings).sample

        # perform guidance
        noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)
        noise_pred = noise_pred_uncond + guidance_scale *
(noise_pred_text - noise_pred_uncond)

        # compute the previous noisy sample x_t -> x_t-1
        latents = f_scheduler.step(noise_pred, t, latents).prev_sample

    # scale and decode the image latents with vae
    latents = 1 / 0.18215 * latents

    with torch.no_grad():
        image = f_vae.decode(latents).sample

    image = (image / 2 + 0.5).clamp(0, 1)
    image = image.detach().cpu().permute(0, 2, 3, 1).numpy()
    images = (image * 255).round().astype("uint8")
    pil_images = [Image.fromarray(image) for image in images]

    return pil_images
```

```
scheduler.init_noise_sigma
```

```
1.0
```

```
imgs = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, scheduler)
imgs[0]
```

```
<ipython-input-14-a94b7c08074d>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"ba574e1bdfbc4250be70d28595bde028","version_major":2,"vers
ion_minor":0}

Student Section

Now that we have this process setup, you're task is to try different VAEs, text encoders, schedulers, etc.. in order to find what you consider the "best" or most photorealistic images that follow the prompt. Your images will be graded using a metric commonly used in generative AI research: FID. You will also be graded on the quality and clarity of your writeup.

Here's an example where run the default vae but with the commonly used DDIM Scheduler:

0. DDIM Scheduler

```
from diffusers import DDIMScheduler, HeunDiscreteScheduler,
LMSDiscreteScheduler, PNDMScheduler

ddim_sched = DDIMScheduler.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="scheduler")
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, ddim_sched, num_steps=50) # Replace the scheduler with
our DDIM
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"7c1cc0cba00142f59f53918447aa0781","version_major":2,"vers
ion_minor":0}

A full list of components available from the diffusers package can be found here: https://huggingface.co/docs/diffusers/index. Some potential schedulers:

- PNDM scheduler (used by default).
- K-LMS scheduler.
- Heun Discrete scheduler.
- DPM Solver Multistep scheduler.

Try **at least 3 new combinations** along with different combinations of hyperparameters such as steps, guidance scale, etc.. and write 2-3 paragraphs about what works well, why it seems to work well, and what you're overall best pipeline is. Try a variety of prompts and **include at least 3 comparison images** in your write-up.

Also include **2 images generated using your best technique for each these prompts**:

- "a big bruin bear roaring at a trojan warrior"
- "a person taking a picture of a landscape"
- "a tall building on a busy street".

Experimenting with different schedulers

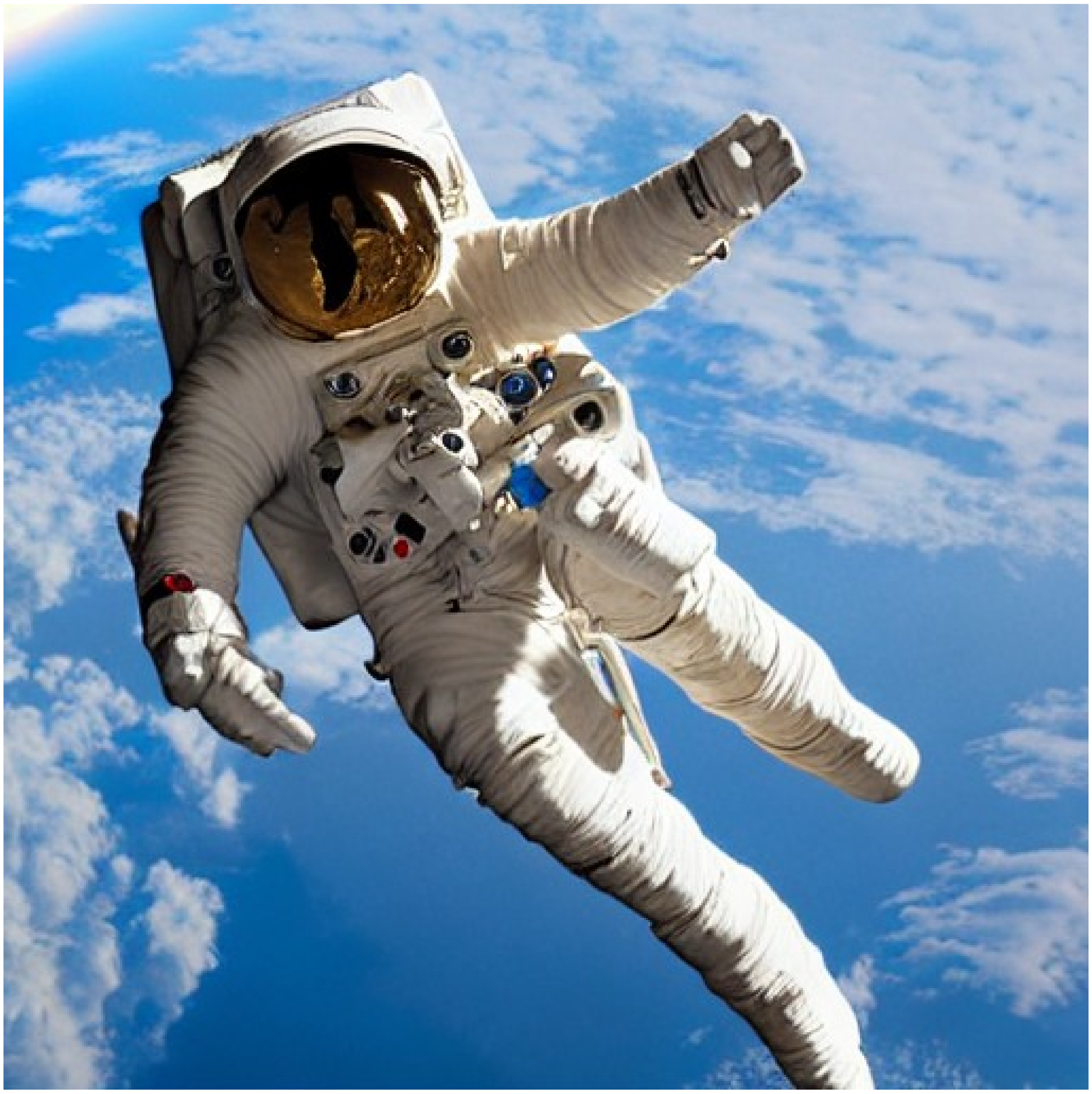
1. DPM Solver Multistep scheduler

1.1Vary num_steps

```
## Student Experimentation here

# Results with DPM Solver Multistep Scheduler

from diffusers import DPMSolverMultistepScheduler

dpm_sched =
DPMSolverMultistepScheduler.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="scheduler")
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=50)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"cb6e934d05cb44d7a009ade2a40d3de0","version_major":2,"vers
ion_minor":0}



```
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=200)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

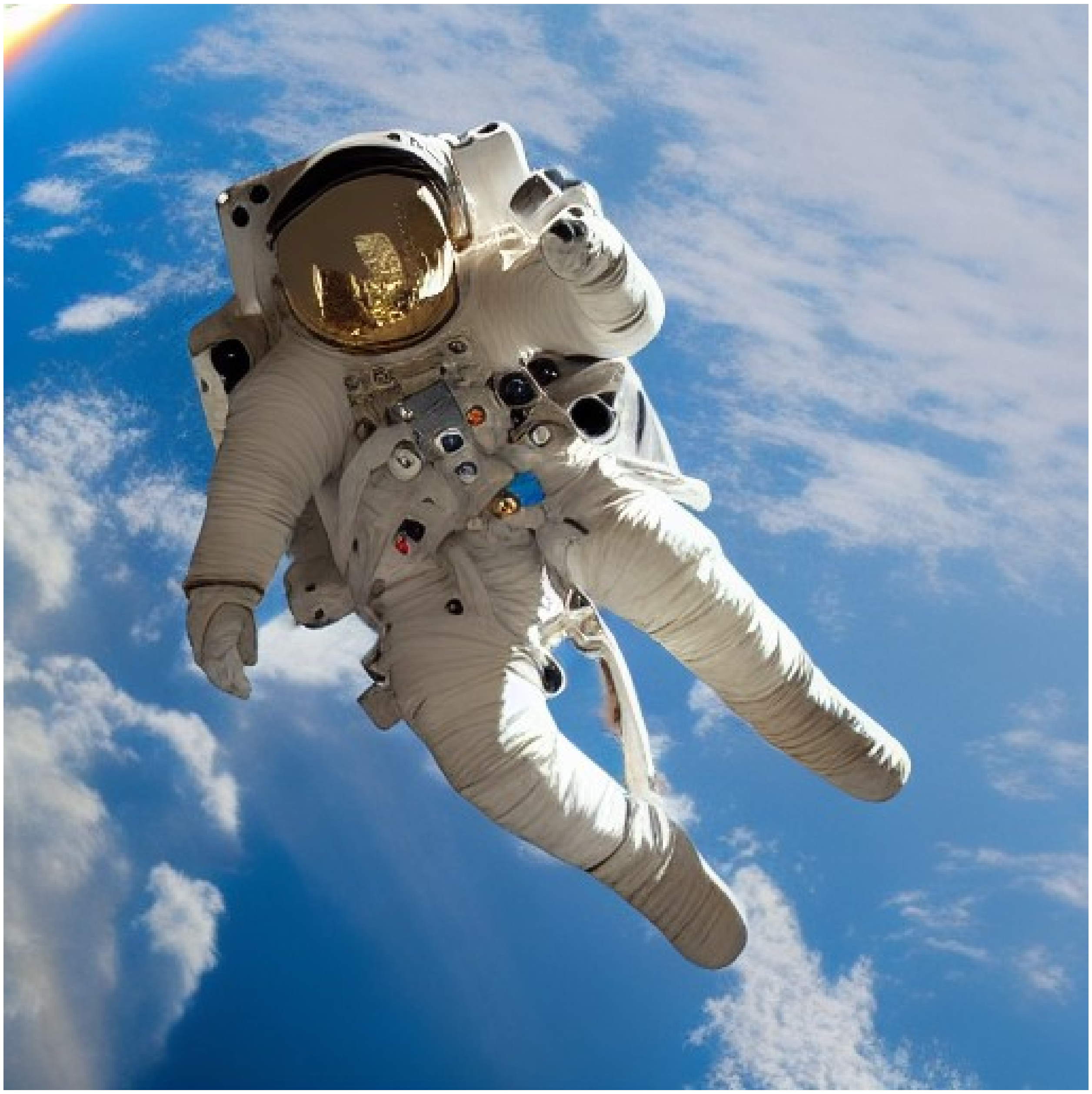
```
{"model_id":"13e94c617fbc42e387ee534cf112fb96","version_major":2,"vers
ion_minor":0}
```

```
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=10)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
```

```
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"b1b4aa1c5b824ed4bf57aca6f37c7d5b","version_major":2,"version_minor":0}

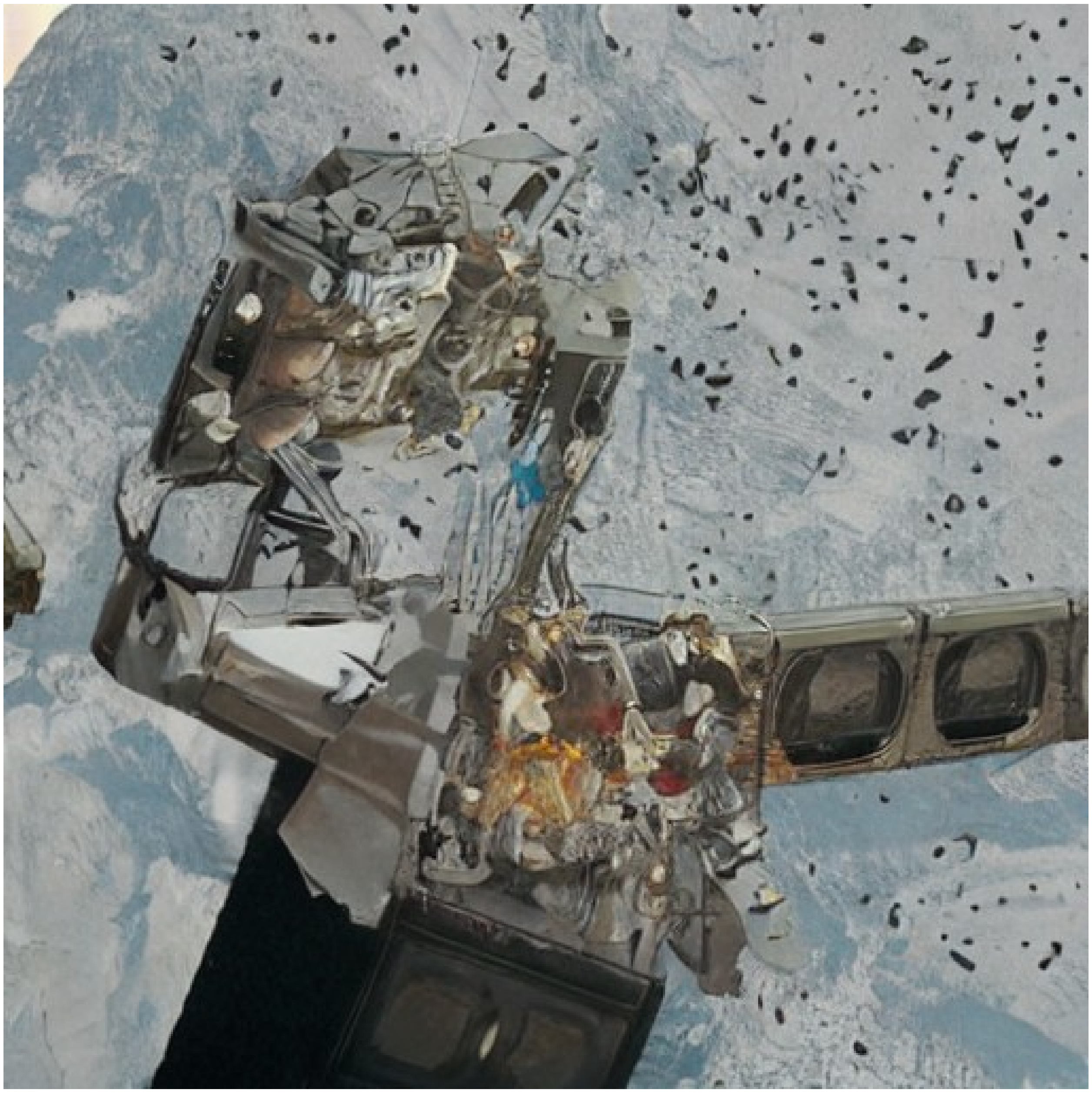


1.2 Vary guidance scale

```
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=50, guidance_scale=1.0)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

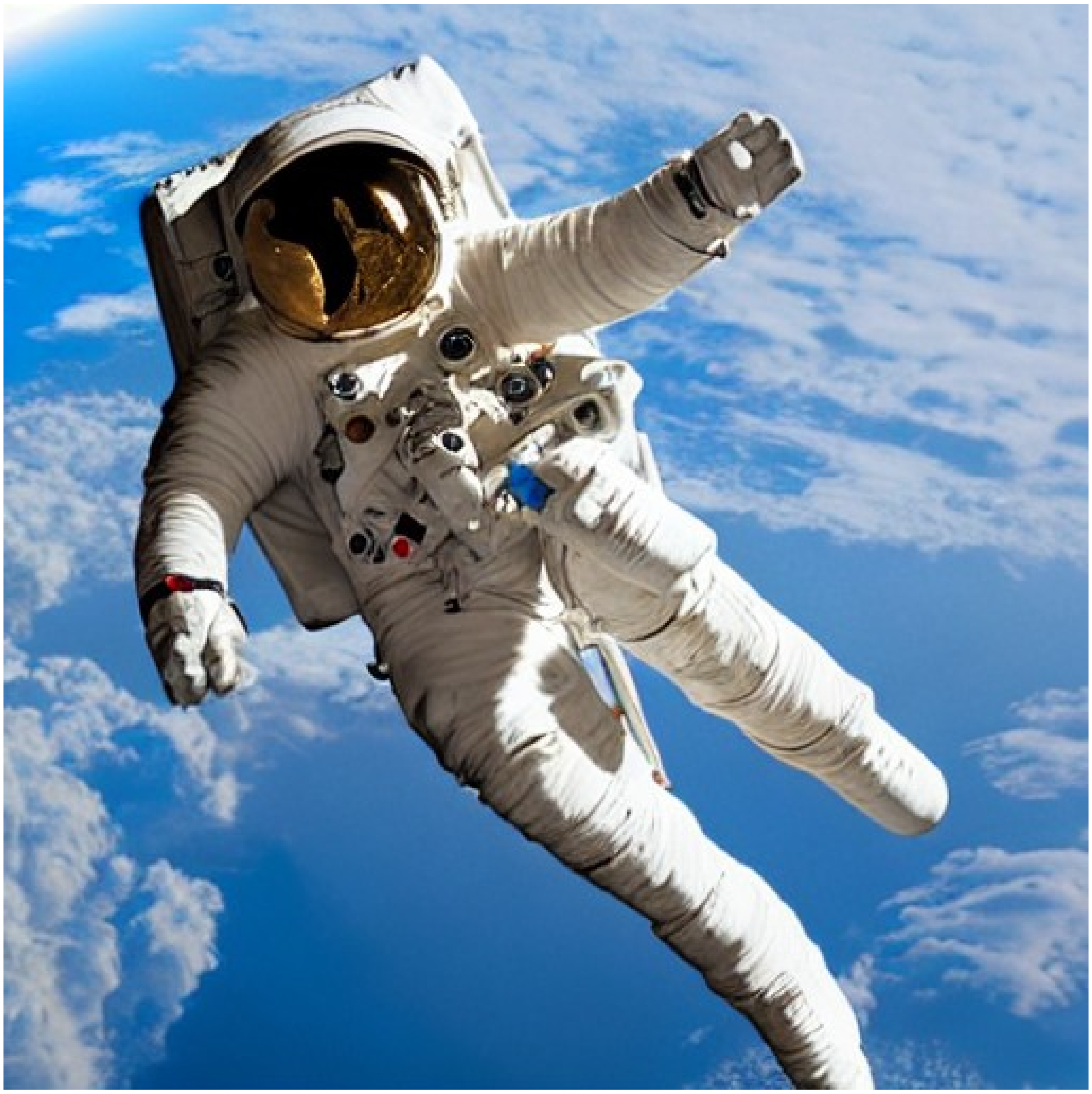
{"model_id":"2737eb6d84f14599ab7178e052732e7d","version_major":2,"version_minor":0}



```
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=50, guidance_scale=8.0)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"860c3c0029b94babb896172fba7a4093","version_major":2,"vers
ion_minor":0}



```
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, dpm_sched, num_steps=50, guidance_scale=25.0)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"194ed192396c4cb9ae58c7bb873bbaa1","version_major":2,"version_minor":0}

2. HEUN scheduler

```
heun_sched = HeunDiscreteScheduler.from_pretrained("CompVis/stable-
diffusion-v1-4", subfolder="scheduler")
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, heun_sched, num_steps=50)
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"39689f77e3fd43258b4bae34a46fdb51","version_major":2,"version_minor":0}

3. LMS scheduler

```
lms_sched = LMSDiscreteScheduler.from_pretrained("CompVis/stable-
diffusion-v1-4", subfolder="scheduler")
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, heun_sched, num_steps=50) # Replace the scheduler with
our DDIM
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"1a24338a69d44baf91ba5f94767fd253","version_major":2,"vers
ion_minor":0}

4. PNDM scheduler

```
pndm_sched = PNDMScheduler.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="scheduler")
img = render_image("an astronaut flying", vae, unet, tokenizer,
text_encoder, pndm_sched, num_steps=50) # Replace the scheduler with
our DDIM
img[0]
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
```

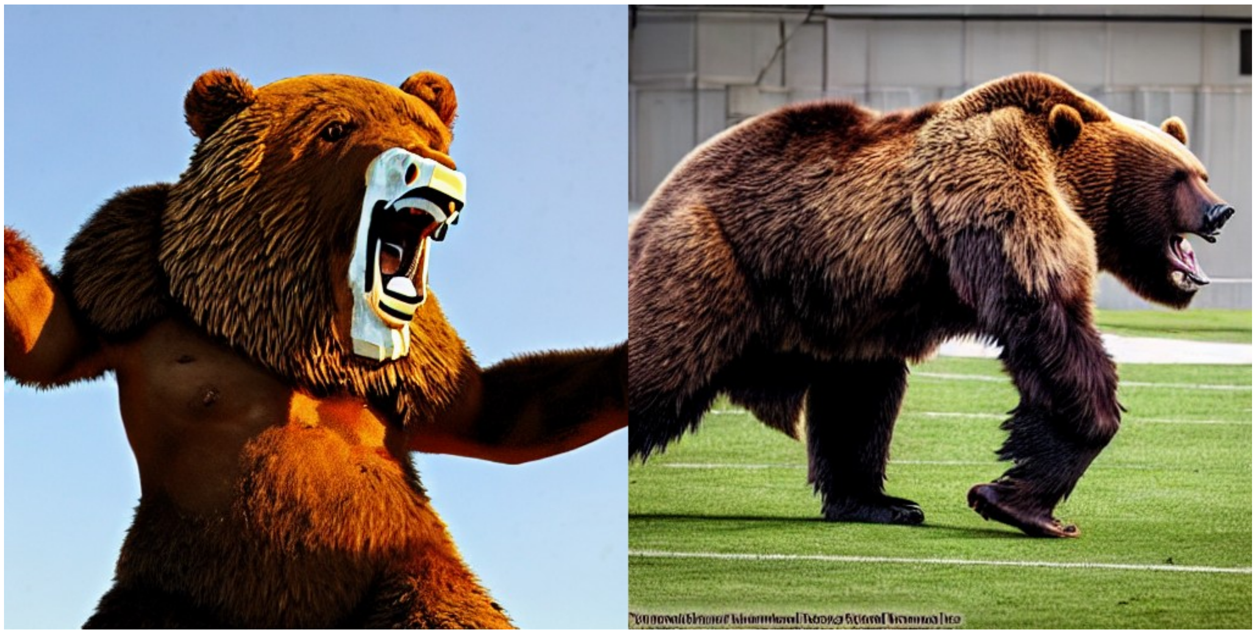
```
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8, 
width // 8), generator=generator)
```

{"model_id":"93bf81db748a4a779888d12017075cb7","version_major":2,"version_minor":0}

Student Images

```
num_images = 2
prompt = ["a big bruin bear roaring at a trojan warrior"] * num_images
seed_value = [42, 70]
```

```
images = [None]*2
for i in range(len(images)):
  images[i] = render_image("a big bruin bear roaring at a trojan
warrior", vae, unet, tokenizer, text_encoder, dpm_sched, num_steps=50,
guidance_scale=7.5, seed = seed_value[i])

grid = image_grid(images, rows=1, cols=2)
grid
```



```
seed_value = [42, 70]
images = [None]*2
for i in range(len(images)):
  images[i] = render_image("a person taking a picture of a landscape",
vae, unet, tokenizer, text_encoder, dpm_sched, num_steps=50,
guidance_scale=7.5, seed = seed_value[i])

grid = image_grid(images, rows=1, cols=2)
grid
```

```
<ipython-input-5-c0448fe887d1>:28: FutureWarning: Accessing config
attribute `in_channels` directly via 'UNet2DConditionModel' object
attribute is deprecated. Please access 'in_channels' over
'UNet2DConditionModel's config object instead, e.g.
'unet.config.in_channels'.
  latents = torch.randn((batch_size, f_unet.in_channels, height // 8,
width // 8), generator=generator)
```

{"model_id":"35ee518189ee4a1d858107b1758812de","version_major":2,"version_minor":0}

{"model_id":"9c460aa7b6574122b7e9c82d89e2deec","version_major":2,"version_minor":0}

Student Write-up (Images rendered above)

NOTE : Images are rendered in the above cells !

The best use of hyperparameters:

1. guidance scale = 7.5
2. num_steps = 50
3. scheduler = DPMSolverMultistepScheduler++
4. seed = [42,70]

With respect to guidance scale, I observed that too less of a guidance scale leads to an image away from the textual prompt, and too much of it causes it to be rendered in an abstract cartoonish style.

For num_steps, as I increase the number of time steps, the model converges well but the change is mostly pronounced between 10 to 50 as compared to 50 to 100 as noted from the above images

With respect to scheduler, the best one with fastest average run time was DPMSolver Multi-step, whereas PNDM gave visually non-appealing results.

Written Questions

1. At the end of the derivation of ELBO from lecture, we are left with two parts:

$$\log p(x) = E_q\left[\log \frac{p(x,z)}{q(z \vee x)}\right) + \log E_q\left[\frac{q(z \vee x)}{p(z \vee x)}\right)$$

- Which of these (left or right), becomes the KL Divergence term?
- Write out the full equation (including the KL divergence term) for what we would want to maximize. Note you can use LaTeX in colab by adding $$ signs before and after the equation.
- Intuitively explain what the KL Divergence term is measuring

Answer:

1. The term towards the right is KL Divergence (ELBO being left).

2. The full equation:

\begin{aligned} & \log p_\theta(x)=E_{q_\phi}(z \mid x)\left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\right]+D_{K L}\left(q_\phi(z \mid x) | p_\theta(z \mid x)\right) \ & \log p_\theta(x) \geq E_{q_\phi}(z \mid x)\left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\right] \longrightarrow E L B O \ & D_{K L}\left(q_\phi(z \mid x) | p_\theta(z \mid x)\right)= E_{q_\phi}(z \mid x)\left[\log \left(\frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right)\right] \& \& \max E L B O=\log p_\theta(x)-D_{K L}\left(q_\phi(z \mid x) | p_\theta(z \mid x)\right)\ & \end{aligned} In the above equations,both the maximum log likelihood and KL Divergence are intractable. And we know that the KL Divergence cannot be negative, it must be greater than or equal to zero. Upon inspecting the equation, we see that maximizing ELBO is the same as minimizing the KL divergence between the two distributions. As our goal is to minimize KL divergence, we would want to maximize ELBO.

1. Intuitively, it quantifies the difference between two probability distributions by measuring the information lost when the second distribution is used to approximate the first one. It sort of measures the average amount of information needed to specify which event occurred based on the second distribution compared to the optimal encoding based on the first distribution. $p_\theta(z \mid x)$, the posterior, is the groud truth. We are approximating the posterior with another distribution, $q_\phi(z \mid x)$.Our goal is to minimize the divergence between the approximation and the posterior so that the approximation is as close as possible.

---

1. We've written out part of the derviation for ELBO for a heirarchical VAE/diffusion model below. Identify the mistakes in these steps, if there are any, and write the correct equation. If there are no mistakes, put "No mistakes"

$$\log p(x) = E_q \left[\log \frac{p(x, z_{1:T})}{q(z_{1:T} \vee x)} \right)$$

$$\log p(x) = E_q \left[\log \frac{p(z_T) p(x \vee z_1) \Pi_{t=2}^{T} p(z_t \vee z_{t-1})}{q(z_1 \vee x) \Pi_{t=2}^{T} q(z_{t-1} \vee z_t)} \right)$$

Answer:

The above equation is wrong.

We assign the distribution $p\left(z_{t-1}, z_t\right) $ to the process of reverse diffusion (denoising), for an arbitary *t* and the distribution $q\left(z_{t}, z_{t-1}\right) $ to the process of forward diffusion. Using Markov property, each node depends on it's neighbouring node in the following way:

\begin{aligned} & p\left(x, z_{1: T}\right)=p\left(z_T\right) p_\theta\left(x \mid z_1\right) \prod_{t=2}^T p_\theta\left(z_{t-1} \mid z_t\right) \ & q\left(z_{1: T} \mid x\right)=q_\phi\left(z_1 \mid x\right) \prod_{t=2}^T q_\phi\left(z_t \mid z_{t-1}\right) \ & \log p(x)=E_q\left[\log \frac{p\left(z_T\right) p_\theta\left(x \mid z_1\right) \prod_{t=2}^T p_\theta\left(z_{t-1} \mid z_t\right)}{q_\phi\left(z_1 \mid x\right) \prod_{t=2}^T q_\phi\left(z_{t} \mid z_{t-1}\right)}\right]\end{aligned}