CS 180 Project 5A

In this problem set you will play around with diffusion models, implement diffusion sampling loops, and use them for other tasks such as inpainting and creating optical illusions.

Part 0: Setup

Using DeepFloyd

We are going to use the DeepFloyd IF diffusion model. DeepFloyd is a two stage model trained by Stability AI. The first stage produces images of size 64×64 and the second stage takes the outputs of the first stage and generates images of size 256×256. We provide upsampling code at the very end of the notebook, though this is not required in your submission.

Before using DeepFloyd, you must accept its usage conditions. To do so:

1. Make a Hugging Face account and log in.
2. Accept the license on the model card of DeepFloyd/IF-I-XL-v1.0. For affiliation, you can fill in "The University of California, Berkeley." Accepting the license on the stage I model card will auto accept for the other IF models.
3. Log in locally by entering your Hugging Face Hub access token below. You should be able to find and create tokens here. A read token is enough for this project.

```
from huggingface_hub import login

token = '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'
login(token=token)
```

Install Dependencies

Run the below to install dependencies.

```
! pip install -q \
  diffusers \
  transformers \
  safetensors \
  sentencepiece \
  accelerate \
  bitsandbytes \
  einops \
  mediapy \
  accelerate
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 59.1/59.1 MB 43.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 67.8 MB/s eta 0:00:00
```

Import Dependencies

The cell below imports useful packages we will need, and setup the device that we're using.

```
from PIL import Image
import mediapy as media
from pprint import pprint
from tqdm import tqdm

import torch
import torchvision.transforms.functional as TF
import torchvision.transforms as transforms
from diffusers import DiffusionPipeline
from transformers import T5EncoderModel

# For downloading web images
import requests
from io import BytesIO

device = 'cuda'
```

```
Flax classes are deprecated and will be removed in Diffusers v1.0.0. We recommend migrating to PyTorch classes or
Flax classes are deprecated and will be removed in Diffusers v1.0.0. We recommend migrating to PyTorch classes or
```

Loading the models

We will need to download and create the two DeepFloyd stages. These models are quite large, so this cell may take a minute or two to run.

```
# Load DeepFloyd IF stage I
stage_1 = DiffusionPipeline.from_pretrained(
    "DeepFloyd/IF-I-L-v1.0",
    text_encoder=None,
    variant="fp16",
    torch_dtype=torch.float16,
)
stage_1.to(device)

# Load DeepFloyd IF stage II
stage_2 = DiffusionPipeline.from_pretrained(
                "DeepFloyd/IF-II-L-v1.0",
                text_encoder=None,
                variant="fp16",
                torch_dtype=torch.float16,
              )
stage_2.to(device)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/t
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
model_index.json: 100%  604/604 [00:00<00:00, 79.0kB/s]

A mixture of fp16 and non-fp16 filenames will be loaded.
Loaded fp16 filenames:
[text_encoder/model.fp16-00002-of-00002.safetensors, text_encoder/model.fp16-00001-of-00002.safetensors, unet/dif
Loaded non-fp16 filenames:
[watermarker/diffusion_pytorch_model.safetensors
If this behavior is not expected, please check your folder structure.
Fetching 12 files: 100%  12/12 [00:07<00:00,  1.27it/s]
tokenizer_config.json: 100%  2.54k/2.54k [00:00<00:00, 47.6kB/s]
config.json: 100%  1.63k/1.63k [00:00<00:00, 30.2kB/s]
config.json: 100%  4.57k/4.57k [00:00<00:00, 73.6kB/s]
special_tokens_map.json: 100%  2.20k/2.20k [00:00<00:00, 50.5kB/s]
tokenizer/spiece.model: 100%  792k/792k [00:00<00:00, 1.43MB/s]
safety_checker/model.fp16.safetensors: 100%  608M/608M [00:03<00:00, 203MB/s]
preprocessor_config.json: 100%  518/518 [00:00<00:00, 39.9kB/s]
scheduler_config.json: 100%  454/454 [00:00<00:00, 39.2kB/s]
watermarker/diffusion_pytorch_model.safe(…): 100%  15.5k/15.5k [00:00<00:00, 37.5kB/s]
config.json: 100%  74.0/74.0 [00:00<00:00, 8.54kB/s]
unet/diffusion_pytorch_model.fp16.safete(…): 100%  1.87G/1.87G [00:07<00:00, 90.8MB/s]
Loading pipeline components...: 100%  6/6 [00:00<00:00,  9.88it/s]
`torch_dtype` is deprecated! Use `dtype` instead!
You are using the default legacy behaviour of the <class 'transformers.models.t5.tokenization_t5.T5Tokenizer'>. T
model_index.json: 100%  692/692 [00:00<00:00, 86.9kB/s]

A mixture of fp16 and non-fp16 filenames will be loaded.
Loaded fp16 filenames:
[text_encoder/model.fp16-00002-of-00002.safetensors, text_encoder/model.fp16-00001-of-00002.safetensors, unet/dif
Loaded non-fp16 filenames:
[watermarker/diffusion_pytorch_model.safetensors
If this behavior is not expected, please check your folder structure.
Fetching 13 files: 100%  13/13 [00:07<00:00,  1.33it/s]
config.json: 100%  4.92k/4.92k [00:00<00:00, 246kB/s]
scheduler_config.json: 100%  424/424 [00:00<00:00, 13.9kB/s]
scheduler_config.json: 100%  454/454 [00:00<00:00, 15.4kB/s]
preprocessor_config.json: 100%  518/518 [00:00<00:00, 10.2kB/s]
tokenizer/spiece.model: 100%  792k/792k [00:00<00:00, 494kB/s]
special_tokens_map.json: 100%  2.20k/2.20k [00:00<00:00, 64.1kB/s]
safety_checker/model.fp16.safetensors: 100%  608M/608M [00:01<00:00, 630MB/s]
```

Play with the Model using Your Own Text Prompts!

DeepFloyd was trained as a text-to-image model, which takes text prompts as input and outputs images that are aligned with the text. However, a raw text string cannot be directly used as the model's input — we first need to convert it into a high-dimensional vector (of 4096 dimensions in our case) that the model can understand, a.k.a. prompt embeddings.

Since prompt encoders are always very big and hard to run in your notebook, we provide two HuggingFace clusters A and B for generating your own prompt embeddings! Both are the same and feel free to use either of them. Please follow the instructions to create a dictionary of embeddings for your prompts, download the resulting `.pth` file, and load it in Google Colab.

Please note that both clusters have daily usage limits, so if you're unable to use one, please try another or try again tomorrow. Alternatively, **START EARLY** and download the `.pth` file in advance — you only need to generate it once, and you can reuse the downloaded file afterward. If you are running out of time, you can download one of our precomputed embeddings, but this is a predefined set of prompts and lacks flexibility. We want to see your creativity!

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Loads your own .pth file here
prompt_embeds_dict = torch.load('/content/drive/MyDrive/cs180_project5_data/prompt_embeds_dict.pth')

# If you want our predefined embeddings, please uncomment the following two lines and run this cell.
# !wget https://cal-cs180.github.io/fa24/hw/proj5/prompt_embeds_dict.pth -O prompt_embeds_dict.pth
# prompt_embeds_dict = torch.load('prompt_embeds_dict.pth')

print("You have the embeddings for the following prompts")
pprint(list(prompt_embeds_dict.keys()))
```

```
You have the embeddings for the following prompts
['a giant library floating in the clouds, glowing with jellyfish lights',
 'a neon samurai standing in a rainy cyberpunk street',
 'a realistic cat dressed like a king in a Renaissance painting',
 'a crystal dragon coming out of a frozen waterfall at sunrise',
 'an astronaut walking inside a huge ancient temple on an alien planet',
 'a tiny house floating inside a glowing glass bubble',
 'a giant whale swimming through a sky full of clouds',
 'a robot painting a sunset on a canvas',
 'a tree growing on top of a small island floating in space',
 'a traditional Chinese ink painting of mountains, water, and a small boat',
 'a cartoon owl sitting on a branch',
 'a small lantern glowing in the night',
 'a realistic portrait of a human face',
 'a small cottage on a hill at sunset',
 'a high quality photo',
 'a rocket ship',
 'a Quinjet',
 'a pyramid',
 'an oil painting of an old man',
 'an oil painting of people around a campfire',
 'a cat',
 'a dog',
 'a landscape of a mountain range',
 'a skull',
 'a waterfull',
 '']
```

By default, we've automatically added the prompt `"a high quality photo"` and the empty prompt `""` to your dictionary. We want you to think of them as "null" prompts that don't have any specific meaning, and are simply a way for the model to do unconditional generation. You can view them as using the diffusion model to "force" a noisy image onto the "manifold" of real images. We will introduce them in a more detailed way in later sections.

Seed your Work

To reproduce your code, please use a random seed from this point onward. Running two runtimes with the same seed will give you identical generation results.

However, the outputs also depend on how many times you run each cell in each runtime, which is harder to keep track of than the seed. Therefore, please **immediately download and save any images you like**, as you may not be able to reproduce them once the notebook runtime expires.

```
def seed_everything(seed):
  torch.cuda.manual_seed(seed)
  torch.manual_seed(seed)

YOUR_SEED = 180
seed_everything(YOUR_SEED)
```

Sampling from the Model

The objects instantiated above, `stage_1` and `stage_2`, already contain code to allow us to sample images using these models. Read the code below carefully (including the comments) and then run the cell to generate some images. Play around with different prompts and `num_inference_steps`.

Deliverables

- Come up with some interesting text prompts and generate their embeddings.
- Choose 3 of your prompts to generate images and display the caption and the output of the model. Briefly reflect on the quality of the outputs and their relationships to the text prompts. Try a different `num_inference_steps` for at least one of the images, showing before and after. Make sure to try at least 2 different `num_inference_steps` values.
- Report the random seed you used. You should use the same seed for all subsequent parts.

Hint

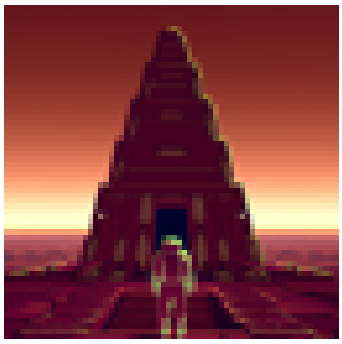
- Since we ask you to generate Visual Anagrams in part 1.8, you may want to include several pairs prompting visual anagrams beforehand.

```
# Get prompt embeddings from the precomputed cache.
# `prompt_embeds` is of shape [N, 77, 4096]
# 77 comes from the max sequence length that deepfloyd will take
# and 4096 comes from the embedding dimension of the text encoder
# `negative_prompt_embeds` is the same shape as `prompt_embeds` and is used
# for Classifier Free Guidance. You can find out more from:
#   - https://arxiv.org/abs/2207.12598
#   - https://sander.ai/2022/05/26/guidance.html
prompts = ["a tiny house floating inside a glowing glass bubble",
           "an astronaut walking inside a huge ancient temple on an alien planet",
           "a giant whale swimming through a sky full of clouds"

]
prompt_embeds = torch.cat([
    prompt_embeds_dict[prompt] for prompt in prompts
], dim=0)
negative_prompt_embeds = torch.cat(
    [prompt_embeds_dict['']] * len(prompts)
)

# Sample from stage 1
# Outputs a [N, 3, 64, 64] torch tensor
# num_inference_steps is an integer between 1 and 1000, indicating how many
# denoising steps to take: lower is faster, at the cost of reduced quality
stage_1_output = stage_1(
    prompt_embeds=prompt_embeds,
    negative_prompt_embeds=negative_prompt_embeds,
    num_inference_steps=50,
    output_type="pt"
).images

# Sample from stage 2
# Outputs a [N, 3, 256, 256] torch tensor
# num_inference_steps is an integer between 1 and 1000, indicating how many
# denoising steps to take: lower is faster, at the cost of reduced quality
stage_2_output = stage_2(
    image=stage_1_output,
    num_inference_steps=50,
    prompt_embeds=prompt_embeds,
    negative_prompt_embeds=negative_prompt_embeds,
    output_type="pt",
).images

# Display images
# We need to permute the dimensions because `media.show_images` expects
# a tensor of shape [N, H, W, C], but the above stages gives us tensors of
# shape [N, C, H, W]. We also need to normalize from [-1, 1], which is the
# output of the above stages, to [0, 1]
media.show_images(
    stage_1_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts, height = 250)
media.show_images(
    stage_2_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts, height = 250)
```

```
100%                                                  50/50 [00:16<00:00,  2.41it/s]

100%                                                  50/50 [01:11<00:00,  1.43s/it]
```

a tiny house floating inside a glowing glass bubble

an astronaut walking inside a huge ancient temple on an alien planet



a giant whale swimming through a sky full of clouds

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.12/dist-packages/PIL/Image.py in fromarray(obj, mode)
   3307         try:
-> 3308             mode, rawmode = _fromarray_typemap[typekey]
   3309         except KeyError as e:

KeyError: ((1, 1), '<f2')

The above exception was the direct cause of the following exception:

TypeError                                 Traceback (most recent call last)
                                  6 frames
/usr/local/lib/python3.12/dist-packages/PIL/Image.py in fromarray(obj, mode)
   3310             typekey_shape, typestr = typekey
   3311             msg = f"Cannot handle this data type: {typekey_shape}, {typestr}"
-> 3312             raise TypeError(msg) from e
   3313     else:
   3314         deprecate("'mode' parameter", 13)

TypeError: Cannot handle this data type: (1, 1), <f2
```

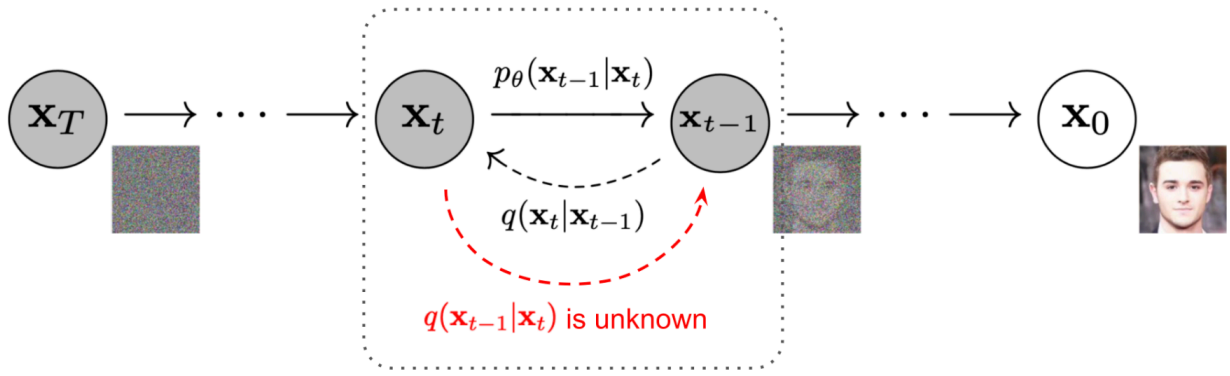
```
import matplotlib.pyplot as plt

# Prepare images: (B, C, H, W) → (B, H, W, C)
images = (
    stage_2_output
    .permute(0, 2, 3, 1)
    .cpu()
    .float()        # ensure float32 for matplotlib
    / 2 + 0.5
).clamp(0, 1).numpy()

# Plot
plt.figure(figsize=(9, 3))   # width scaled for 3 images
for i in range(images.shape[0]):
    plt.subplot(1, images.shape[0], i + 1)
    plt.imshow(images[i])
    plt.axis('off')

plt.show()
```

Part 1: Sampling Loops

In this part of the problem set, you will write your own "sampling loops" that use the pretrained DeepFloyd denoisers. These should produce high quality images such as the ones generated above.

You will then modify these sampling loops to solve different tasks such as inpainting or producing optical illusions.

Diffusion Models Primer



(Image Source)

Starting with a clean image, x_0, we can iteratively add noise to an image, obtaining progressively more and more noisy versions of the image, x_t, until we're left with basically pure noise at timestep $t = T$. When $t = 0$, we have a clean image, and for larger t more noise is in the image.

A diffusion model tries to reverse this process by denoising the image. By giving a diffusion model a noisy x_t and the timestep t, the model predicts the noise in the image. With the predicted noise, we can either completely remove the noise from the image, to obtain an estimate of x_0, or we can remove just a portion of the noise, obtaining an estimate of x_{t-1}, with slightly less noise.

To generate images from the diffusion model (sampling), we start with pure noise at timestep T sampled from a gaussian distribution, which we denote x_T. We can then predict and remove part of the noise, giving us x_{T-1}. Repeating this process until we arrive at x_0 gives us a clean image.

For the DeepFloyd models, $T = 1000$.

Setup

The exact amount of noise added at each step is dictated by noise coefficients, $\bar{\alpha}_t$, which were chosen by the people who trained DeepFloyd. Run the cell below to create `alphas_cumprod`, which retrieves these coefficients. The number of coefficients is the same as the number of sampling steps.

```
# Get scheduler parameters
alphas_cumprod = stage_1.scheduler.alphas_cumprod
print(f"We have in total {alphas_cumprod.shape[0]} noise coefficients")
```

```
We have in total 1000 noise coefficients
```

We provide you our favorite Campanilie as a test image that you can work with!

```
# Get test image
!wget cal-cs180.github.io/fa24/hw/proj5/assets/campanile.jpg -O campanile.jpg
test_im = Image.open('campanile.jpg')

# For stage 1: Resize to (64, 64), convert to tensor, rescale to [-1, 1], and
# add a batch dimension. The result is a (1, 3, 64, 64) tensor
test_im = Image.open('campanile.jpg').resize((64, 64))
test_im = TF.to_tensor(test_im)
test_im = 2 * test_im - 1
test_im = test_im[None]

# Show test image
print('Test image:')
media.show_image(test_im[0].permute(1,2,0) / 2. + 0.5)
```

```
--2025-12-08 20:34:27--  http://cal-cs180.github.io/fa24/hw/proj5/assets/campanile.jpg
Resolving cal-cs180.github.io (cal-cs180.github.io)... 185.199.108.153, 185.199.109.153, 185.199.110.153, ...
Connecting to cal-cs180.github.io (cal-cs180.github.io)|185.199.108.153|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://cal-cs180.github.io/fa24/hw/proj5/assets/campanile.jpg [following]
--2025-12-08 20:34:27--  https://cal-cs180.github.io/fa24/hw/proj5/assets/campanile.jpg
Connecting to cal-cs180.github.io (cal-cs180.github.io)|185.199.108.153|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 75608 (74K) [image/jpeg]
Saving to: 'campanile.jpg'

campanile.jpg       100%[===================>]  73.84K  --.-KB/s    in 0.01s

2025-12-08 20:34:28 (5.94 MB/s) - 'campanile.jpg' saved [75608/75608]

Test image:
```

1.1 Implementing the forward process

Disclaimer about equations: Colab sometimes cannot correctly render the math equations below. Please cross-reference them with the part A webpage to make sure that you're looking at the fully correct equation.

A key part of diffusion is the forward process, which takes a clean image and adds noise to it. In this part, we will write a function to implement this. The forward process is defined by:

$$q(x_t|x_0) = N(x_t; \sqrt{\bar{\alpha}_t}\, x_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{1}$$

which is equivalent to computing

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon \quad \text{where } \epsilon \sim N(0,1) \tag{2}$$

That is, given a clean image x_0, we get a noisy image x_t at timestep t by sampling from a Gaussian with mean $\sqrt{\bar{\alpha}_t}\, x_0$ and variance $(1-\bar{\alpha}_t)$. Note that the forward process is not *just* adding noise -- we also scale the image.

You will need to use the `alphas_cumprod` variable, which contains the $\bar{\alpha}_t$ for all $t \in [0, 999]$. Remember that $t = 0$ corresponds to a clean image, and larger t corresponds to more noise. Thus, $\bar{\alpha}_t$ is close to 1 for small t, and close to 0 for large t. Run the forward process on the test image with $t \in [250, 500, 750]$. Show the results -- you should get progressively more noisy images.

Delivarables

- Implement the `im_noisy = forward(im, t)` function
- Show the Campanile at noise level [250, 500, 750]

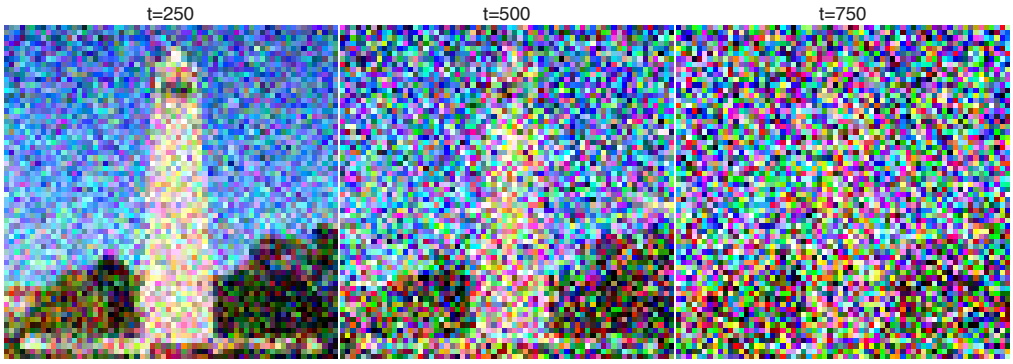
Hints

- The `torch.randn_like` function is helpful for computing ϵ.
- Use the `alphas_cumprod` variable, which contains an array of the hyperparameters, with `alphas_cumprod[t]` corresponding to $\bar{\alpha}_t$.
- This is notebook, to show images, we recommend you to use `media.show_image` function. Documentation available here.

```
def forward(im, t):
  """
  Args:
    im : torch tensor of size (1, 3, 64, 64) representing the clean image
    t : integer timestep

  Returns:
    im_noisy : torch tensor of size (1, 3, 64, 64) representing the noisy image at timestep t
  """
  with torch.no_grad():
    # ===== your code here! ====
    alpha_bar_t = alphas_cumprod[t].to(im.device).type_as(im)
    eps = torch.randn_like(im)
    im_noisy = torch.sqrt(alpha_bar_t) * im + torch.sqrt(1 - alpha_bar_t) * eps

    # ===== end of code ====
  return im_noisy
```

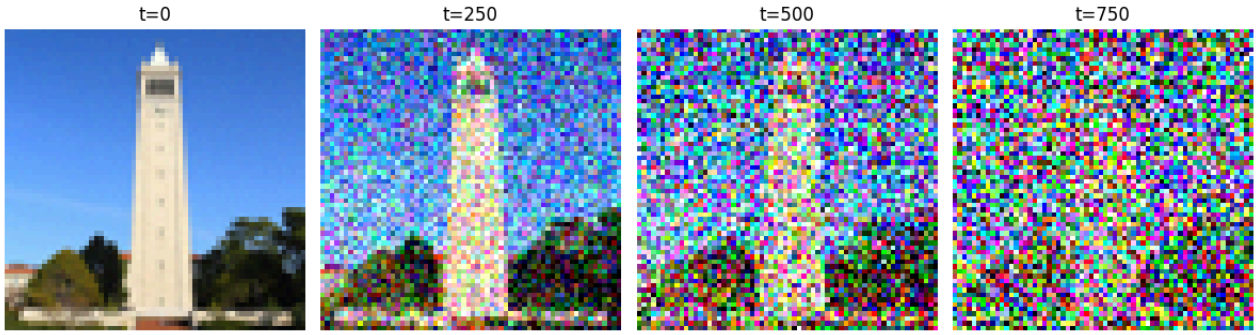
```
# Show the test image at noise level [250, 500, 750]
# ===== your code here! ====
timesteps = [250, 500, 750]
noisy_images = []

for t in timesteps:
```

```
    im_noisy = forward(test_im, t)
    # Convert from [-1,1] to [0,1]
    im_disp = im_noisy[0].permute(1,2,0) / 2 + 0.5
    noisy_images.append(im_disp)

media.show_images(noisy_images, titles=[f"t={t}" for t in timesteps], height = 250)
# ===== end of code ====
```




```
import matplotlib.pyplot as plt
levels = [0, 250, 500, 750]

fig, axs = plt.subplots(1, len(levels), figsize=(12, 4))

for i, t in enumerate(levels):
    x_t = forward(test_im, t)
    img = x_t[0].permute(1,2,0).cpu() / 2 + 0.5

    axs[i].imshow(img)
    axs[i].set_title(f"t={t}")
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
```



1.2 Classical Denoising

Let's try to denoise these images using classical methods. Again, take noisy images for timesteps [250, 500, 750], but use **Gaussian blur filtering** to try to remove the noise. Getting good results should be quite difficult, if not impossible.

Deliverables

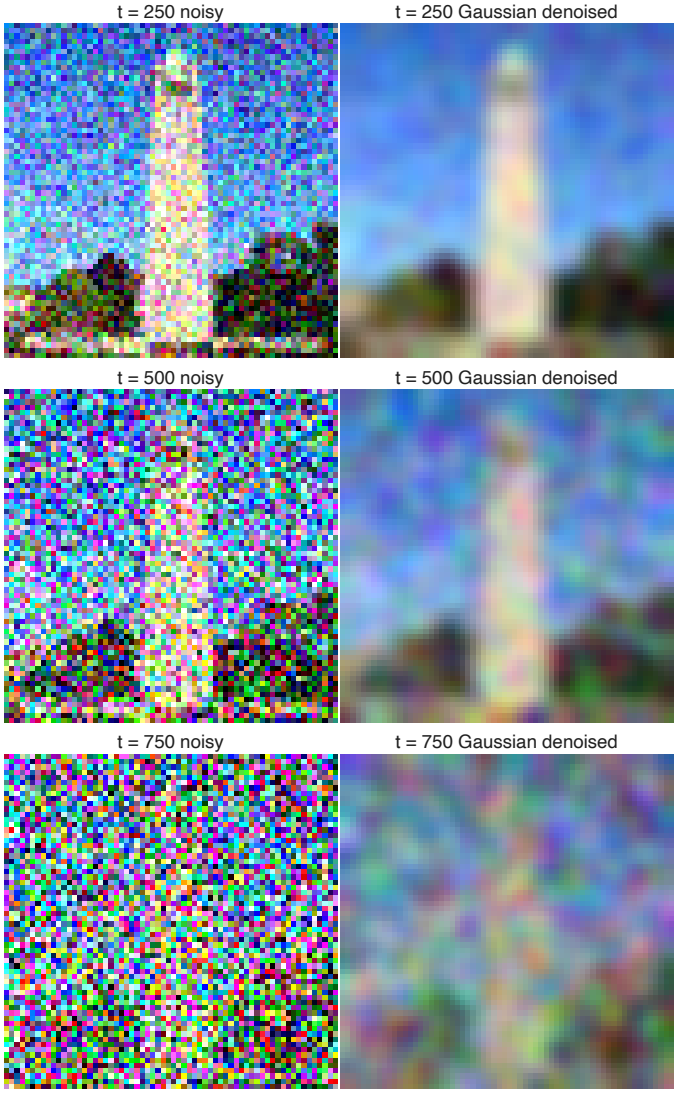
- For each of the 3 noisy Campanile images from the previous part, show your best Gaussian-denoised version side by side.

Hint

- `torchvision.transforms.functional.gaussian_blur` is useful. Here is the documentation.

```
# ===== your code here! ====
timesteps = [250, 500, 750]
```

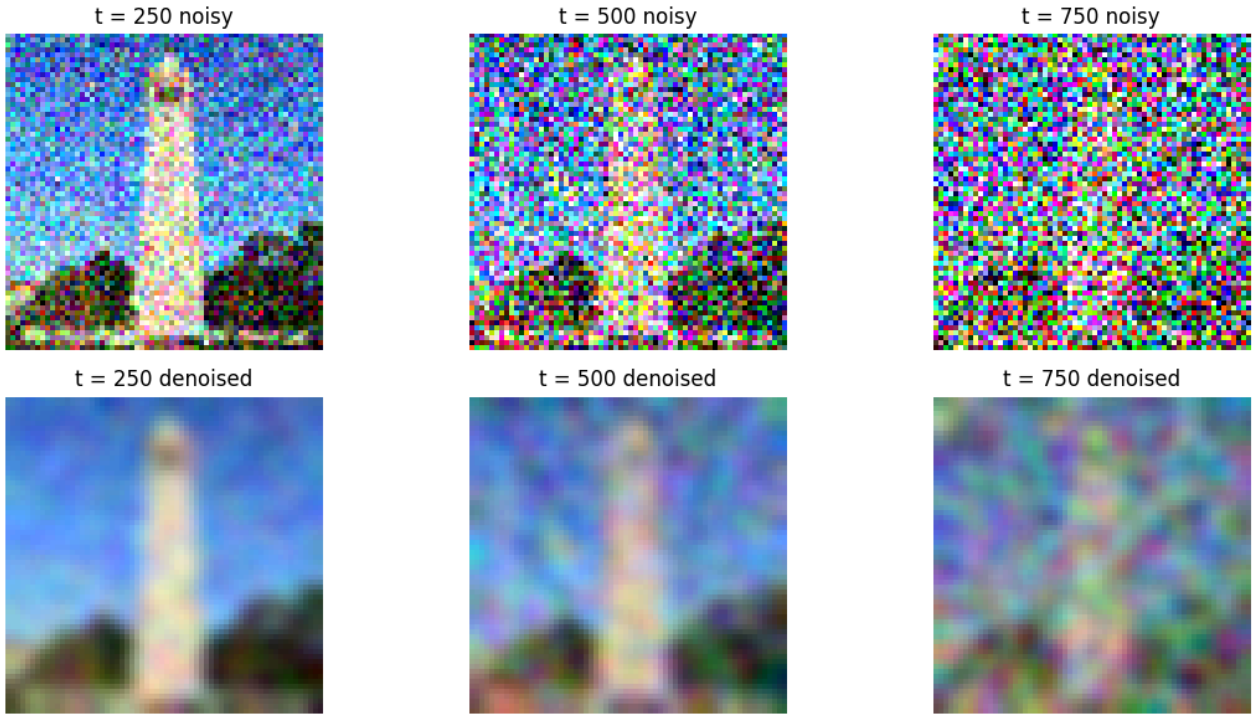
```
for t in timesteps:
    im_noisy = forward(test_im, t)
    im_denoised = TF.gaussian_blur(im_noisy, kernel_size=7, sigma=2.0)
    noisy_disp = (im_noisy[0].permute(1,2,0) / 2 + 0.5).cpu()
    denoised_disp = (im_denoised[0].permute(1,2,0) / 2 + 0.5).cpu()
    media.show_images(
        [noisy_disp, denoised_disp],
        titles=[f"t = {t} noisy", f"t = {t} Gaussian denoised"],
        height = 250
    )
# ===== end of code ====
```




```
import matplotlib.pyplot as plt

timesteps = [250, 500, 750]

# 2 rows (noisy, denoised) × 3 columns (timesteps)
fig, axs = plt.subplots(2, len(timesteps), figsize=(12, 6))

noisy_images = []
denoised_images = []

# Generate images first
for t in timesteps:
    im_noisy = forward(test_im, t)
    im_denoised = TF.gaussian_blur(im_noisy, kernel_size=7, sigma=2.0)

    noisy_images.append((im_noisy[0].permute(1,2,0) / 2 + 0.5).cpu())
    denoised_images.append((im_denoised[0].permute(1,2,0) / 2 + 0.5).cpu())

# Plot noisy images (top row)
for i, (t, img) in enumerate(zip(timesteps, noisy_images)):
    axs[0, i].imshow(img)
    axs[0, i].set_title(f"t = {t} noisy")
    axs[0, i].axis("off")

# Plot denoised images (bottom row)
```

```
for i, (t, img) in enumerate(zip(timesteps, denoised_images)):
    axs[1, i].imshow(img)
    axs[1, i].set_title(f"t = {t} denoised")
    axs[1, i].axis("off")

plt.tight_layout()
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
```



1.3 Implementing One Step Denoising

Now, we'll use a pretrained diffusion model to denoise. The actual denoiser can be found at `stage_1.unet`. This is a UNet that has already been trained on a *very, very* large dataset of (x_0, x_t) pairs of images. We can use it to recover Gaussian noise from the image. Then, we can remove this noise to recover (something close to) the original image. Note: this UNet is conditioned on the amount of Gaussian noise by taking timestep t as additional input.

Because this diffusion model was trained with text conditioning, we also need a text prompt embedding. We provide the embedding for the prompt `"a high quality photo"` for you to use. Later on, you can use your own text prompts.

Deliverables

For the 3 noisy images from 1.2 (t = [250, 500, 750]):

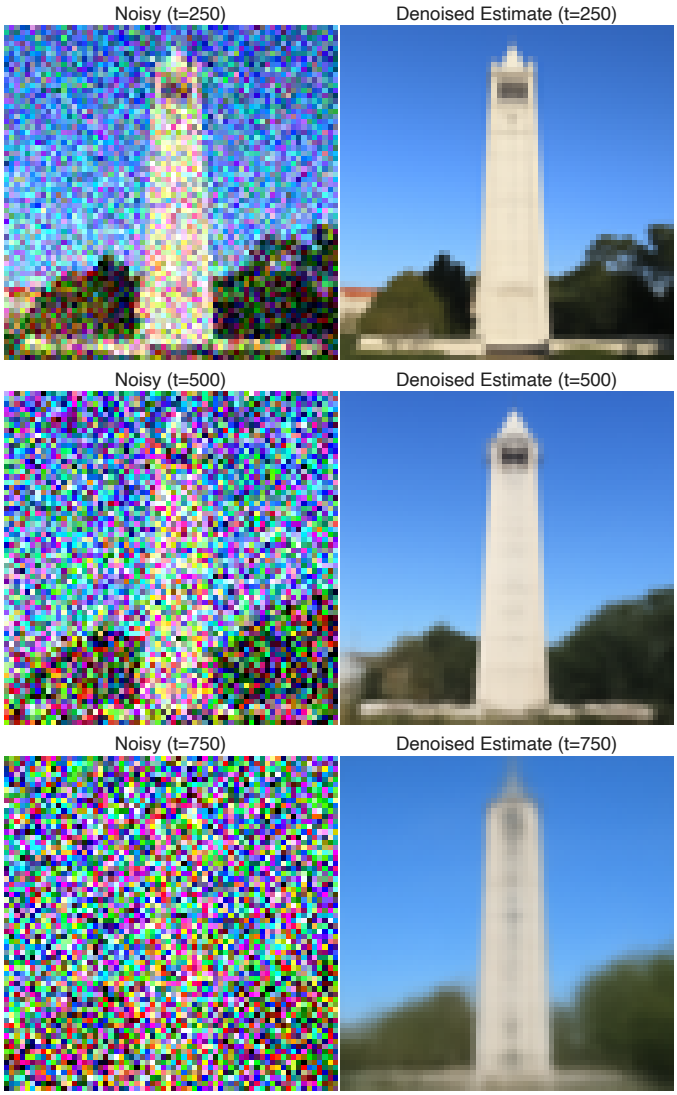
- Use your `forward` function to add noise to your Campanilie
- Estimate the noise in the new noisy image, by passing it through `stage_1.unet`
- Remove the noise from the noisy image to obtain an estimate of the original image
- Visualize the original image, the noisy image, and the estimate of the original image

Hints

- When removing the noise, you can't simply subtract the noise estimate. Recall that in equation 2 we need to scale the noise. Look at equation 2 to figure out how we predict x_0 from x_t and t.
- You will probably have to wrangle tensors to the correct device and into the correct data types. The functions `.to(device)` and `.half()` will be useful. The denoiser is loaded on the device `cuda` as `half` precision (to save memory), so inputs to the denoiser need to match them.
- The signature for the unet is `stage_1.unet(im_noisy, t, encoder_hidden_states=prompt_embeds, return_dict=False)`. You need to pass in the noisy image, the timestep, and the prompt embeddings. The `return_dict` argument just makes the output nicer.

- The unet will output a tensor of shape (1, 6, 64, 64). This is because DeepFloyd was trained to predict the noise as well as variance of the noise. The first 3 channels is the noise estimate, which you will use. The second 3 channels is the variance estimate which you may ignore for now.
- To save GPU memory, you should wrap all of your code in a `with torch.no_grad():` context. This tells torch not to do automatic differentiation, and saves a considerable amount of memory.

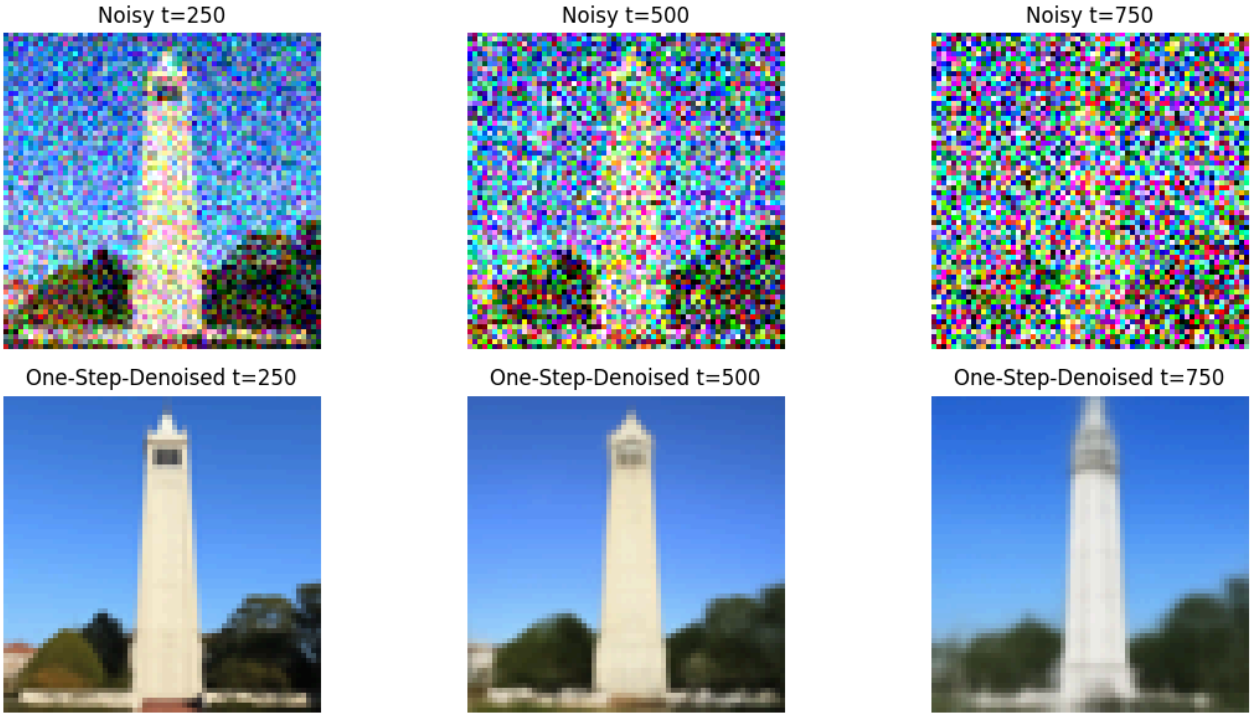
```
# Please use the null prompt embedding
prompt_embeds = prompt_embeds_dict[""].half().to(device)

with torch.no_grad():
  for t in [250, 500, 750]:
    # Get alpha bar
    alpha_bar = alphas_cumprod[t]   # scalar

    # Run forward process
    # ===== your code here! =====

    # Generate noisy image x_t from x_0
    im_noisy = torch.sqrt(alpha_bar) * test_im + torch.sqrt(1 - alpha_bar) * torch.randn_like(test_im)

    # ==== end of code ====

    # Estimate noise in noisy image
    noise_est = stage_1.unet(
        im_noisy.half().cuda(),
        t,
        encoder_hidden_states=prompt_embeds,
        return_dict=False
    )[0]

    # Take only first 3 channels, and move result to cpu
    noise_est = noise_est[:, :3].cpu()

    # Remove the noise (estimate x0)
    # ===== your code here! =====

    # x0_hat = (x_t - sqrt(1 - alpha_bar) * eps_hat) / sqrt(alpha_bar)
    x0_hat = (im_noisy - torch.sqrt(1 - alpha_bar) * noise_est) / torch.sqrt(alpha_bar)

    # For display: convert to numpy in [0,1]
    x0_hat_disp = (x0_hat[0].permute(1,2,0).numpy() / 2 + 0.5)

    # ==== end of code ====

    # Show results
    media.show_images(
        [
            (im_noisy[0].permute(1,2,0).cpu() / 2 + 0.5),
            x0_hat_disp
        ],
        titles=[ f"Noisy (t={t})", f"Denoised Estimate (t={t})"],
        height=250
    )
```




```
import matplotlib.pyplot as plt

timesteps = [250, 500, 750]

noisy_list = []
denoised_list = []

prompt_embeds = prompt_embeds_dict[""].half().to(device)

with torch.no_grad():
    for t in timesteps:
        alpha_bar = alphas_cumprod[t]

        im_noisy = torch.sqrt(alpha_bar) * test_im + torch.sqrt(1 - alpha_bar) * torch.randn_like(test_im)

        noise_est = stage_1.unet(
            im_noisy.half().cuda(),
            t,
            encoder_hidden_states=prompt_embeds,
            return_dict=False
        )[0]

        noise_est = noise_est[:, :3].cpu()

        x0_hat = (im_noisy - torch.sqrt(1 - alpha_bar) * noise_est) / torch.sqrt(alpha_bar)

        noisy_disp = (im_noisy[0].permute(1,2,0).cpu() / 2 + 0.5).numpy()
        denoised_disp = (x0_hat[0].permute(1,2,0).cpu() / 2 + 0.5).numpy()

        noisy_list.append(noisy_disp)
        denoised_list.append(denoised_disp)

fig, axs = plt.subplots(2, len(timesteps), figsize=(12, 6))

for i, (t, img) in enumerate(zip(timesteps, noisy_list)):
    axs[0, i].imshow(img)
    axs[0, i].set_title(f"Noisy t={t}")
```

```
        axs[0, i].axis("off")

    for i, (t, img) in enumerate(zip(timesteps, denoised_list)):
        axs[1, i].imshow(img)
        axs[1, i].set_title(f"One-Step-Denoised t={t}")
        axs[1, i].axis("off")

    plt.tight_layout()
    plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0
```



1.4 Implementing Iterative Denoising

In part 1.3, you should see that the denoising UNet does a much better job of projecting the image onto the natural image manifold, but it does get worse as you add more noise. This makes sense, as the problem is much harder with more noise!

But diffusion models are designed to denoise iteratively. In this part we will implement this.

In theory, we could start with noise x_{1000} at timestep $T = 1000$, denoise for one step to get an estimate of x_{999}, and carry on until we get x_0. But this would require running the diffusion model 1000 times, which is quite slow (and costs $$$).

It turns out, we can actually speed things up by skipping steps. The rationale for why this is possible is due to a connection with differential equations. It's a tad complicated, and out of scope for this course, but if you're interested you can check out this excellent article.

To skip steps we can create a list of timesteps that we'll call `strided_timesteps`, which will be much shorter than the full list of 1000 timesteps. `strided_timesteps[0]` will correspond to the the largest t (and thus the noisiest image) and `strided_timesteps[-1]` will correspond to $t = 0$ (and thus a clean image). One simple way of constructing this list is by introducing a regular stride step (e.g. stride of 30 works well).

On the `i`th denoising step we are at $t =$ `strided_timesteps[i]`, and want to get to $t' =$ `strided_timesteps[i+1]` (from more noisy to less noisy). To actually do this, we have the following formula:

$$x_{t'} = \frac{\sqrt{\bar{\alpha}_{t'}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t'})}{1-\bar{\alpha}_t} x_t + v_\sigma \tag{3}$$

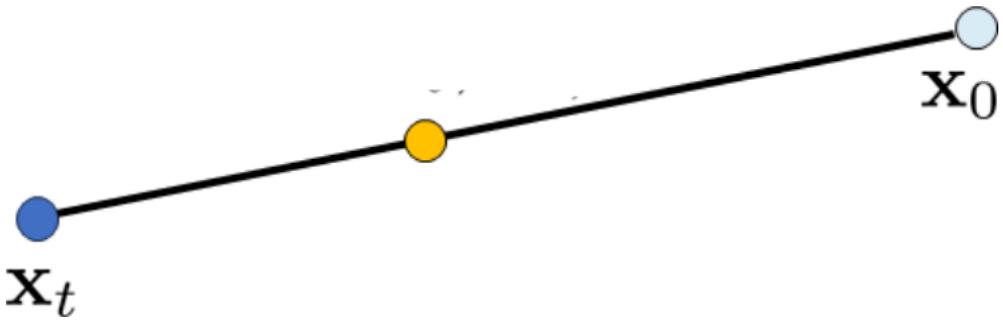
where:

- x_t is your image at timestep t
- $x_{t'}$ is your noisy image at timestep t' where $t' < t$ (less noisy)
- $\bar{\alpha}_t$ is defined by `alphas_cumprod`, as explained above.

- $\alpha_t = \bar{\alpha}_t / \bar{\alpha}_{t'}$
- $\beta_t = 1 - \alpha_t$
- x_0 is our current estimate of the clean image c (you can copy past from the previous section)

The v_σ is random noise, which in the case of DeepFloyd is also predicted. The process to compute this is not very important for us, so we supply a function, `add_variance`, to do this for you.

You can think of this as a linear interpolation between the signal and noise:



(Image Source)

For more information, see equations 6 and 7 of the DDPM paper for more information (Denoising Diffusion Probabilistic Models, the paper that introduces the diffusion model, which comes from Cal!). Be careful about bars above the alpha! Some have them and some do not.

First create the list `strided_timesteps`. You should start at timestep 990, and take step sizes of size 30 until you arrive at 0. After completing the problem set, feel free to try different "schedules" of timesteps.

Also implement the function `iterative_denoise(im_noisy, i_start)`, which takes a noisy image `im_noisy`, as well as a starting index `i_start`. The function should denoise an image starting at timestep `timestep[i_start]`, applying the above formula to obtain an image at timestep `t' = timestep[i_start + 1]`, and repeat iteratively until we arrive at a clean image.

Add noise to the Campanilie test image to timestep `timestep[10]` and display this image. Then run the `iterative_denoise` function on the noisy image, with `i_start = 10`, to obtain a clean image and display it. Please display every 5th image of the denoising loop. Compare this to the "one-step" denoising method from the previous section, and to gaussian blurring.

Deliverables

Using `i_start = 10`:

- Create `strided_timesteps`: a list of monotonically decreasing timesteps, starting at 990, with a stride of 30, eventually reaching 0. Also initialize the timesteps using the function `stage_1.scheduler.set_timesteps(timesteps=strided_timesteps)`
- Complete the `iterative_denoise` function
- Show the noisy Campanilie every 5th loop of denoising (it should gradually become less noisy)
- Show the final predicted clean image, using iterative denoising
- Show the predicted clean image using only a single denoising step, as was done in the previous part. This should look much worse.
- Show the predicted clean image using gaussian blurring, as was done in part 1.2.

Hints

- Remember, the unet will output a tensor of shape (1, 6, 64, 64). This is because DeepFloyd was trained to predict the noise as well as variance of the noise. The first 3 channels is the noise estimate, which you will use here. The second 3 channels is the variance estimate which you will pass to the `add_variance` function
- Read the documentation for the `add_variance` function to figure out how to use it to add the v_σ to the image.
- Depending on if your final images are torch tensors or numpy arrays, you may need to modify the `show_images` call a bit.

```
# Make timesteps. Must be list of ints satisfying:
# - monotonically decreasing
# - ends at 0
# - begins close to or at 999

# create `strided_timesteps`, a list of timesteps, from 990 to 0 in steps of 30
# ===== your code here! =====
strided_timesteps = []
```

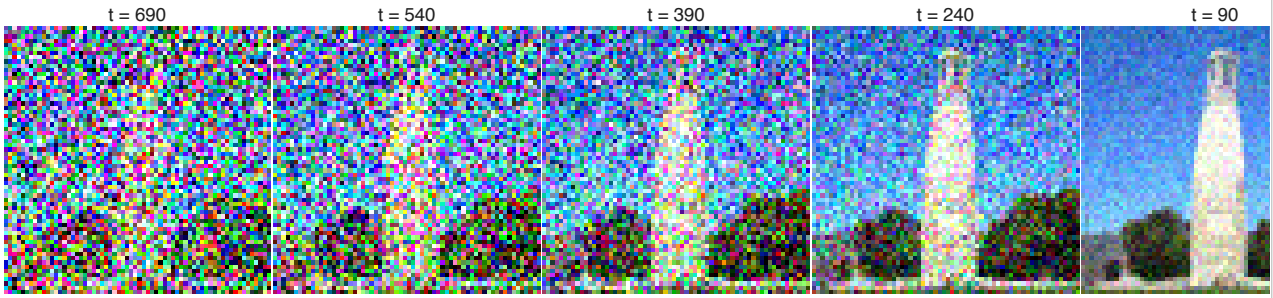
```
i = 990
while i >= 0:
  strided_timesteps.append(i)
  i = i - 30

# ==== end of code ====

stage_1.scheduler.set_timesteps(timesteps=strided_timesteps)   # Need this b/c variance computation
```

```
def add_variance(predicted_variance, t, image):
  '''
  Args:
    predicted_variance : (1, 3, 64, 64) tensor, last three channels of the UNet output
    t: scale tensor indicating timestep
    image : (1, 3, 64, 64) tensor, noisy image

  Returns:
    (1, 3, 64, 64) tensor, image with the correct amount of variance added
  '''
  # Add learned variance
  variance = stage_1.scheduler._get_variance(t, predicted_variance=predicted_variance)
  variance_noise = torch.randn_like(image)
  variance = torch.exp(0.5 * variance) * variance_noise
  return image + variance
```

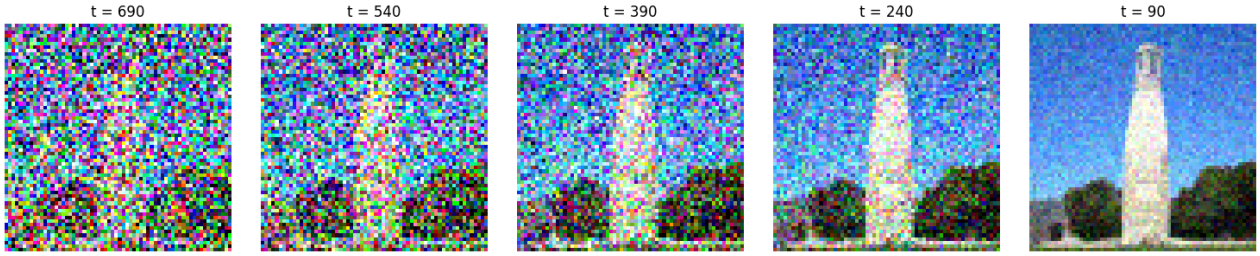
```
def iterative_denoise(im_noisy, i_start, prompt_embeds, timesteps, display=True):
    image = im_noisy
    images_every5 = []

    with torch.no_grad():
        for i in range(i_start, len(timesteps) - 1):

            # Save every 5th image
            if (i - i_start) % 5 == 0:
                img_disp = (image[0].permute(1, 2, 0)/2 + 0.5).cpu().detach()
                images_every5.append(img_disp)

            # Timesteps
            t = timesteps[i]
            prev_t = timesteps[i+1]

            # Alpha and beta
            alpha_cumprod = alphas_cumprod[t]
            alpha_cumprod_prev = alphas_cumprod[prev_t]
            alpha = alpha_cumprod / alpha_cumprod_prev
            beta = 1 - alpha

            # UNet prediction
            model_output = stage_1.unet(
                image,
                t,
                encoder_hidden_states=prompt_embeds,
                return_dict=False
            )[0]

            noise_est, predicted_variance = torch.split(model_output, image.shape[1], dim=1)

            # x0 estimate
            x0_est = (image - torch.sqrt(1 - alpha_cumprod) * noise_est) / torch.sqrt(alpha_cumprod)

            # DDPM update
            pred_prev_image = (
                (torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cumprod) * x0_est +
                (torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - alpha_cumprod) * image
            )

            pred_prev_image = add_variance(predicted_variance, t, pred_prev_image)

            image = pred_prev_image

        clean = (image[0].permute(1, 2, 0)/2 + 0.5).cpu().detach()

    return clean, images_every5
```

```
i_start = 10
t = strided_timesteps[i_start]
im_noisy = forward(test_im, t )

clean, images_every5 = iterative_denoise(
    im_noisy.half().to(device),
```

```
    i_start=10,
    prompt_embeds=prompt_embeds.half().to(device),
    timesteps=strided_timesteps
)

titles = [f"t = {strided_timesteps[10 + 5*k]}"
          for k in range(len(images_every5))]
media.show_images(images_every5, titles=titles, height=200)
```




```
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, len(images_every5), figsize=(3 * len(images_every5), 3))

for i, (img, title) in enumerate(zip(images_every5, titles)):
    # ---- Convert to numpy float32 safely ----
    if isinstance(img, torch.Tensor):
        img = img.detach().cpu().float().numpy()   # ensures float32
    elif hasattr(img, "astype"):
        img = img.astype(np.float32)
    else:
        img = np.array(img, dtype=np.float32)

    # Clip to valid range [0,1] for matplotlib
    img = np.clip(img, 0, 1)

    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

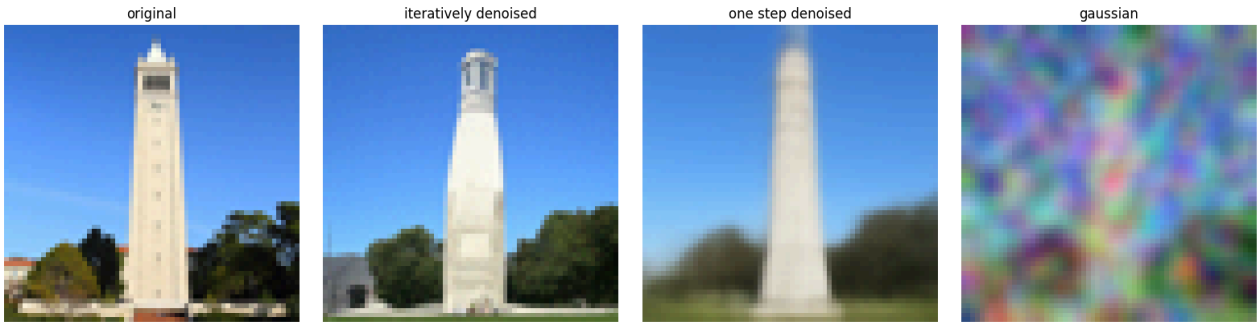



```
media.show_image(clean)
```

```
gaussian_denoised = TF.gaussian_blur(im_noisy, kernel_size=7, sigma=2.0)
gaussian_denoised = gaussian_denoised[0].permute(1, 2, 0)/2 + 0.5

original = test_im[0].permute(1, 2, 0)/2 + 0.5

with torch.no_grad():
    for t in timesteps:
        alpha_bar = alphas_cumprod[t]
        im_noisy = torch.sqrt(alpha_bar) * test_im + torch.sqrt(1 - alpha_bar) * torch.randn_like(test_im)
        noise_est = stage_1.unet(
            im_noisy.half().cuda(),
            t,
```

```
            encoder_hidden_states=prompt_embeds,
            return_dict=False
        )[0]
        noise_est = noise_est[:, :3].cpu()
        x0_hat = (im_noisy - torch.sqrt(1 - alpha_bar) * noise_est) / torch.sqrt(alpha_bar)
        one_step_denoised = (x0_hat[0].permute(1,2,0).cpu() / 2 + 0.5).numpy()

media.show_images([original, clean, one_step_denoised, gaussian_denoised], titles = [f"original", f"iteratively
```


```
import numpy as np
import matplotlib.pyplot as plt
import torch

def to_img(x):
    # Torch tensor → numpy
    if isinstance(x, torch.Tensor):
        x = x.detach().cpu()
        if x.ndim == 3 and x.shape[0] in [1,3]:   # CHW → HWC
            x = x.permute(1,2,0)
        x = x.float().numpy()

    # numpy but channel-first → convert
    elif isinstance(x, np.ndarray):
        if x.ndim == 3 and x.shape[0] in [1,3]:   # CHW numpy → HWC
            x = np.transpose(x, (1,2,0))
        x = x.astype(np.float32)

    # ensure float32
    x = x.astype(np.float32)

    # Clip for display
    x = np.clip(x, 0, 1)

    return x

# ---- NOW USE IT ----

images = [original, clean, one_step_denoised, gaussian_denoised]
titles = ["original", "iteratively denoised", "one step denoised", "gaussian"]

fig, axs = plt.subplots(1, len(images), figsize=(4 * len(images), 4))

for i, (img, title) in enumerate(zip(images, titles)):
    img = to_img(img)
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



1.5 Diffusion Model Sampling

In part 1.4, we use the diffusion model to denoise an image. Another thing we can do with the `iterative_denoise` function is to generate images from scratch. We can do this by setting `i_start = 0` and passing `im_noisy` as random noise. This effectively denoises pure noise. Please do this, and show 5 results of the prompt `"a high quality photo"`.

Deliverables

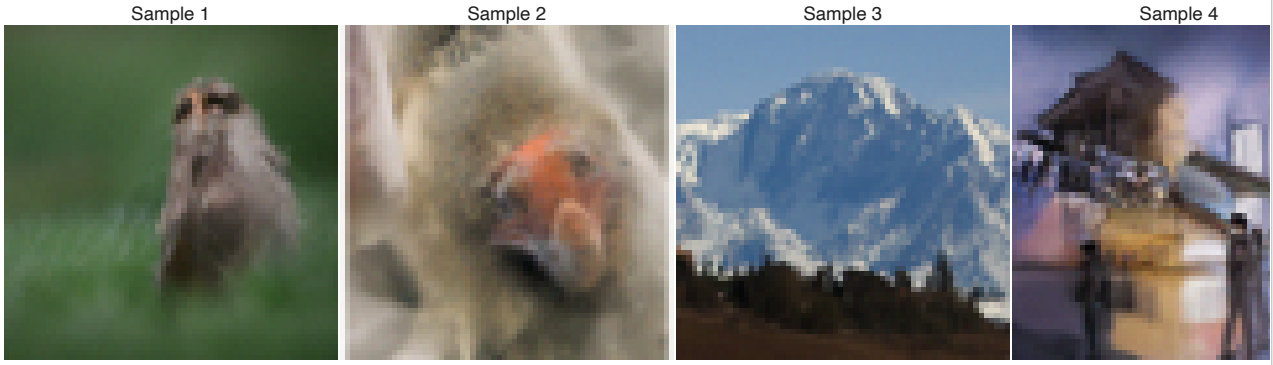
- Show 5 sampled images

Hints

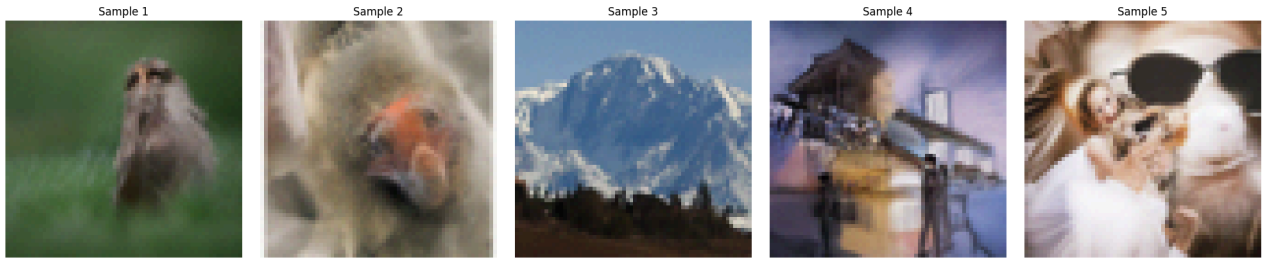
- Use `torch.randn` to make the noise.
- Make sure you move the tensor to the correct device and correct data type by calling `.half()` and `.to(device)`.
- The quality of the images will not be spectacular, but should be reasonable images. We will fix this in the next section with CFG.

```
# Please use this text prompt
prompt_embeds = prompt_embeds_dict["a high quality photo"].half().to(device)

num_samples = 5
samples = []

with torch.no_grad():
    for k in range(num_samples):
        print(f"Sampling image {k+1}/{num_samples}...")
        im_noisy = torch.randn(1, 3, 64, 64).half().to(device)

        clean_sample, _ = iterative_denoise(
            im_noisy,
            i_start=0,
            prompt_embeds=prompt_embeds,
            timesteps=strided_timesteps
        )


        samples.append(clean_sample)

titles = [f"Sample {i+1}" for i in range(num_samples)]

media.show_images(samples, titles=titles, height=250)
```

```
Sampling image 1/5...
Sampling image 2/5...
Sampling image 3/5...
Sampling image 4/5...
Sampling image 5/5...
```




```
import numpy as np
import matplotlib.pyplot as plt
import torch

def to_img(x):
    # Torch tensor → numpy
    if isinstance(x, torch.Tensor):
        x = x.detach().cpu()
        # If CHW, convert to HWC
        if x.ndim == 3 and x.shape[0] in [1, 3]:
            x = x.permute(1, 2, 0)
        x = x.float().numpy()

    # Numpy array
    elif isinstance(x, np.ndarray):
        if x.ndim == 3 and x.shape[0] in [1, 3]:  # CHW case
            x = np.transpose(x, (1, 2, 0))
        x = x.astype(np.float32)

    # Ensure float32 always
    x = x.astype(np.float32)

    # Clip valid range for matplotlib
    x = np.clip(x, 0, 1)

    return x

# ---- Horizontal plot ----
fig, axs = plt.subplots(1, len(samples), figsize=(4 * len(samples), 4))

for i, (img, title) in enumerate(zip(samples, titles)):
    img = to_img(img)
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



1.6 Classifier Free Guidance

You may have noticed that some of the generated images in the prior section are not very good. In order to greatly improve image quality (at the expense of image diversity), we can use a technique called Classifier-Free Guidance.

In CFG, we compute both a noise estimate conditioned on a text prompt, and an unconditional noise estimate. We denote these ϵ_c and ϵ_u. Then, we let our new noise estimate be

$$\epsilon = \epsilon_u + \gamma(\epsilon_c - \epsilon_u) \tag{4}$$

where γ controls the strength of CFG. Notice that for $\gamma = 0$, we get an unconditional noise estimate, and for $\gamma = 1$ we get the conditional noise estimate. The magic happens when $\gamma > 1$. In this case, we get much higher quality images. Why this happens is still up to vigorous debate. For more information on CFG, you can check out this blog post.

Please implement the `iterative_denoise_cfg` function, identical to the `iterative_denoise` function but using classifier-free guidance. To get an unconditional noise estimate, we can just pass an empty prompt embedding to the diffusion model (the model was trained to predict an unconditional noise estimate when given an empty text prompt).

Disclaimer

Before, we used `"a high quality photo"` as a "null" condition. Now, we will use the actual `""` null prompt for unconditional guidance for CFG. In the later part, you should always use `""` null prompt for unconditional guidance.

Deliverables

- Implement the `iterative_denoise_cfg` function
- Show 5 images with a prompt `"a high quality photo"` with a CFG scale of $\gamma = 7$. Now this prompt becomes a **condition** (but fairly weak) to generate **conditional** noise! You will use your customized prompts as stronger conditions in part 1.7 - part 1.9.

Hints

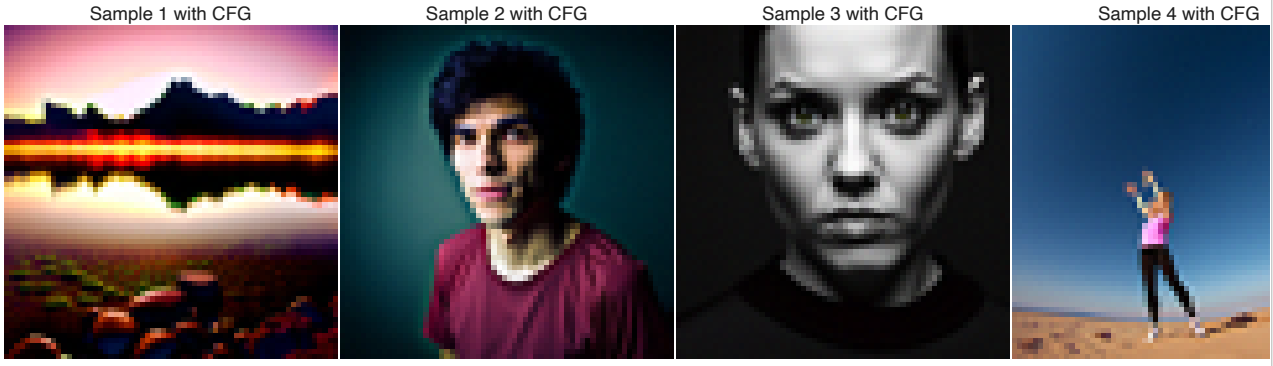
- You will need to run the UNet twice, once for the conditional prompt embedding, and once for the unconditional
- The UNet will predict both a conditional and an unconditional variance. Just use the conditional variance with the `add_variance` function.
- The resulting images should be much better than those in the prior section

```
# The conditional prompt embedding
prompt_embeds = prompt_embeds_dict['a high quality photo'].half().to(device)

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)


def iterative_denoise_cfg(im_noisy, i_start, prompt_embeds, uncond_prompt_embeds,
                          timesteps, scale=7, display=True):
    image = im_noisy

    with torch.no_grad():
        for i in range(i_start, len(timesteps) - 1):


            t = timesteps[i]
            prev_t = timesteps[i + 1]

            alpha_cumprod = alphas_cumprod[t]
            alpha_cumprod_prev = alphas_cumprod[prev_t]
            alpha = alpha_cumprod / alpha_cumprod_prev
            beta = 1 - alpha

            model_output = stage_1.unet(
                image,
                t,
                encoder_hidden_states=prompt_embeds,
                return_dict=False
            )[0]

            uncond_model_output = stage_1.unet(
                image,
                t,
                encoder_hidden_states=uncond_prompt_embeds,
                return_dict=False
            )[0]

            # Split into noise + variance
            noise_c, predicted_variance = torch.split(model_output, image.shape[1], dim=1)
            noise_u, _ = torch.split(uncond_model_output, image.shape[1], dim=1)

            noise_est = noise_u + scale * (noise_c - noise_u)
            x0_est = (image - torch.sqrt(1 - alpha_cumprod) * noise_est) / torch.sqrt(alpha_cumprod)
            pred_prev_image = (
```

```
                (torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cumprod) * x0_est +
                (torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - alpha_cumprod) * image
            )


            pred_prev_image = add_variance(predicted_variance, t, pred_prev_image)
            # ==== end of code ====

            # Step forward
            image = pred_prev_image

        clean = (image[0].permute(1, 2, 0)/2 +0.5).cpu().detach().numpy()

    return clean
```

```
prompt_embeds = prompt_embeds_dict["a high quality photo"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

num_samples = 5
samples = []

with torch.no_grad():
    for k in range(num_samples):
        print(f"Sampling image {k+1}/{num_samples}...")

        im_noisy = torch.randn(1, 3, 64, 64).half().to(device)

        clean_sample = iterative_denoise_cfg(
            im_noisy,
            i_start=0,
            prompt_embeds=prompt_embeds,
            uncond_prompt_embeds=uncond_prompt_embeds,
            timesteps=strided_timesteps,
            scale=7
        )

        samples.append(clean_sample)

titles = [f"Sample {i+1} with CFG" for i in range(num_samples)]
media.show_images(samples, titles=titles, height=250)
```

```
Sampling image 1/5...
Sampling image 2/5...
Sampling image 3/5...
Sampling image 4/5...
Sampling image 5/5...
```




```
fig, axs = plt.subplots(1, len(samples), figsize=(4 * len(samples), 4))

for i, (img, title) in enumerate(zip(samples, titles)):
    img = to_img(img)
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

1.7 Image-to-image Translation

In part 1.4, we take a real image, add noise to it, and then denoise. This effectively allows us to make edits to existing images. The more noise we add, the larger the edit will be. This works because in order to denoise an image, the diffusion model must to some extent "hallucinate" new things -- the model has to be "creative." Another way to think about it is that the denoising process "forces" a noisy image back onto the manifold of natural images.

Here, we're going to take the original Campanile image, noise it a little, and force it back onto the image manifold without any conditioning. Effectively, we're going to get an image that is similar to the Campanile (with a low-enough noise level). This follows the SDEdit algorithm.

To start, please run the forward process to get a noisy Campanile, and then run the `iterative_denoise_cfg` function using a starting index of [1, 3, 5, 7, 10, 20] steps and show the results, labeled with the starting index. You should see a series of "edits" to the original image, gradually matching the original image closer and closer.

Deliverables

- Edits of the Campanile image, using the given prompt at noise levels [1, 3, 5, 7, 10, 20] with the conditional text prompt `"a high quality photo"`
- Edits of 2 of your own test images, using the same procedure.

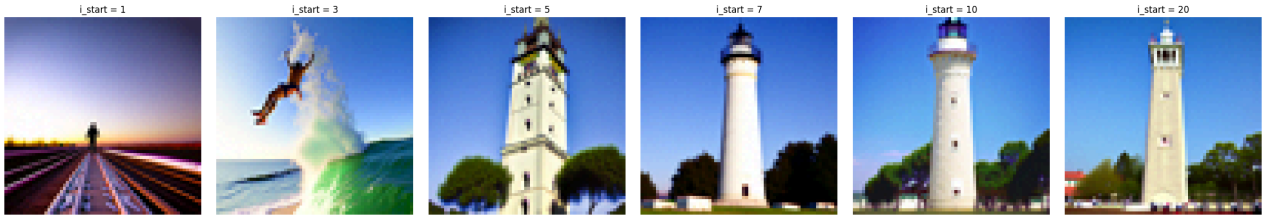
Hints

- You should have a range of images, gradually looking more like the original image

```
prompt_embeds = prompt_embeds_dict['a high quality photo'].half().to(device)

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

indices = [1, 3, 5, 7, 10, 20]
edits = []

for i in indices:
    t = strided_timesteps[i]
    im_noise = forward(test_im, t).half().to(device)

    clean_sample = iterative_denoise_cfg(
        im_noise,
        i_start=i,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edits.append(clean_sample)
```

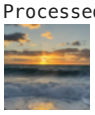
```
titles = [f"i_start = {i}" for i in indices]
media.show_images(edits, titles, height = 250)
```


```
fig, axs = plt.subplots(1, len(edits), figsize=(4 * len(edits), 4))

for i, (img, title) in enumerate(zip(edits, titles)):
    img = to_img(img)   # you already defined this earlier
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```




```
path = "/content/drive/MyDrive/cs180_project5_data/sunrise.jpg"
img = Image.open(path)
sunrise = process_pil_im(img)
```

```
Processed image
```



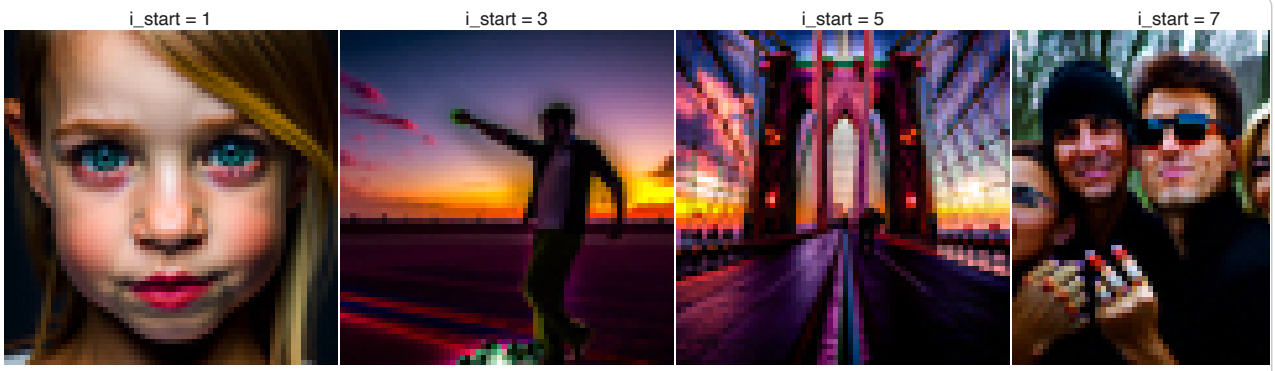
```
prompt_embeds = prompt_embeds_dict['a high quality photo'].half().to(device)

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

indices = [1, 3, 5, 7, 10, 20]
edits = []

for i in indices:
    t = strided_timesteps[i]
    im_noise = forward(sunrise, t).half().to(device)

    clean_sample = iterative_denoise_cfg(
        im_noise,
        i_start=i,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edits.append(clean_sample)
edits.append(sunrise[0].permute(1, 2, 0)/2 + 0.5)
titles = [f"i_start = {i}" for i in indices] + [f"sunrise(original)"]
media.show_images(edits, titles, height = 250)
```


```
fig, axs = plt.subplots(1, len(edits), figsize=(4 * len(edits), 4))

for i, (img, title) in enumerate(zip(edits, titles)):
    img = to_img(img)        # you already defined to_img earlier
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

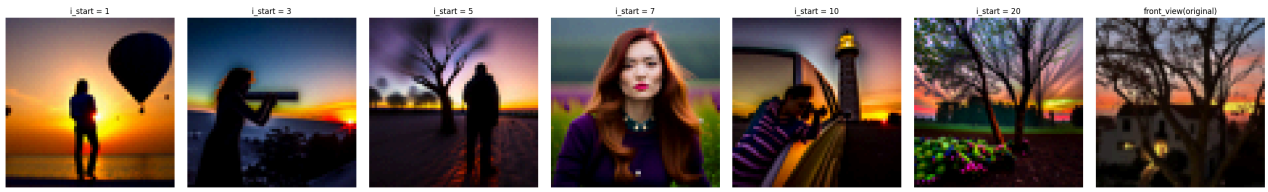

```
path = "/content/drive/MyDrive/cs180_project5_data/front_view.jpg"
front_view = Image.open(path)
front_view = process_pil_im(front_view)
```

```
Processed image
```

```
prompt_embeds = prompt_embeds_dict['a high quality photo'].half().to(device)

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

indices = [1, 3, 5, 7, 10, 20]
edits = []

for i in indices:
    t = strided_timesteps[i]
    im_noise = forward(front_view, t).half().to(device)

    clean_sample = iterative_denoise_cfg(
        im_noise,
        i_start=i,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edits.append(clean_sample)
edits.append(front_view[0].permute(1, 2, 0)/2 + 0.5)
titles = [f"i_start = {i}" for i in indices] + [f"front_view(original)"]
media.show_images(edits, titles, height = 250)
```




```
fig, axs = plt.subplots(1, len(edits), figsize=(4 * len(edits), 4))

for i, (img, title) in enumerate(zip(edits, titles)):
    img = to_img(img)        # you already defined to_img earlier
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



1.7.1 Editing Hand-Drawn and Web Images

This procedure works particularly well if we start with a nonrealistic image (e.g. painting, a sketch, some scribbles) and project it onto the natural image manifold.

Please experiment by starting with hand-drawn or other non-realistic images and see how you can get them onto the natural image manifold in fun ways.

We provide you with 2 ways to provide inputs to the model:

1. download images from the web
2. draw your own images

Please find an image from the internet and apply edits exactly as above. And also draw your own images, and apply edits exactly as above. Feel free to copy the prior cell here. For drawing inspiration, you can check out the examples on this project page.

Deliverables

- 1 image from the web of your choice, edited using the above method for noise levels [1, 3, 5, 7, 10, 20] (and whatever additional noise levels you want)
- 2 hand drawn images, edited using the above method for noise levels [1, 3, 5, 7, 10, 20] (and whatever additional noise levels you want)

Hints

- We provide you with preprocessing code to convert web images to the format expected by DeepFloyd.
- Unfortunately, the drawing interface is hardcoded to be 300x600 pixels, but we need a square image. The code will center crop, so just draw in the middle of the canvas.

Function to Process Images

```
# @title Function to Process Images

def process_pil_im(img):
  '''
  Transform a PIL image
```

```
  '''

  # Convert to RGB
  img = img.convert('RGB')

  # Define the transform to resize, convert to tensor, and normalize to [-1, 1]
  transform = transforms.Compose([
      transforms.Resize(64),               # Resize shortest side to 64
      transforms.CenterCrop(64),             # Center crop
      transforms.ToTensor(),               # Convert image to PyTorch tensor with range [0, 1]
      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize to range [-1, 1]
  ])

  # Apply the transformations and add batch dim
  img = transform(img)[None]

  # Show image
  print("Processed image")
  media.show_image(img[0].permute(1,2,0) / 2 + 0.5)

  return img
```

Download Images from Web

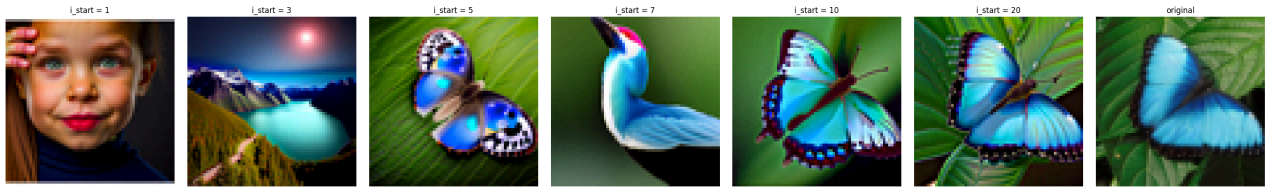
```
# @title Download Images from Web

################
## CHANGE URL ##
################
url = "https://npr.brightspotcdn.com/dims4/default/9743532/2147483647/strip/true/crop/1600x1200+0+0/resize/1760x
################
################

# Download image from URL and process
response = requests.get(url)
web_im = Image.open(BytesIO(response.content))
web_im = process_pil_im(web_im)
```

```
Processed image
```

```
prompt_embeds = prompt_embeds_dict["a high quality photo"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

imgs_list = []
for i_start in tqdm([1, 3, 5, 7, 10, 20]):
  # Add noise
  t = strided_timesteps[i_start]
  im_noisy = forward(web_im, t).half().to(device)

  # Denoise
  clean = iterative_denoise_cfg(im_noisy,
                                i_start=i_start,
                                prompt_embeds=prompt_embeds,
                                uncond_prompt_embeds=uncond_prompt_embeds,
                                timesteps=strided_timesteps,
                                display=False)

  # Add to dict to display later
  imgs_list.append(clean)

imgs_list.append(web_im[0].permute(1, 2, 0)/2 +0.5)
titles = [f"i_start = {i}" for i in [1, 3, 5, 7, 10, 20]] +[f"original"]
media.show_images(imgs_list, titles = titles, height = 250)
```

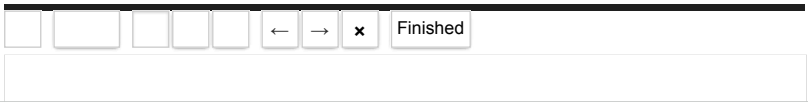
```
100%|██████████| 6/6 [00:23<00:00,  3.97s/it]
```


```
fig, axs = plt.subplots(1, len(imgs_list), figsize=(4 * len(imgs_list), 4))

for i, (img, title) in enumerate(zip(imgs_list, titles)):
    img = to_img(img)      # already defined in your notebook
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



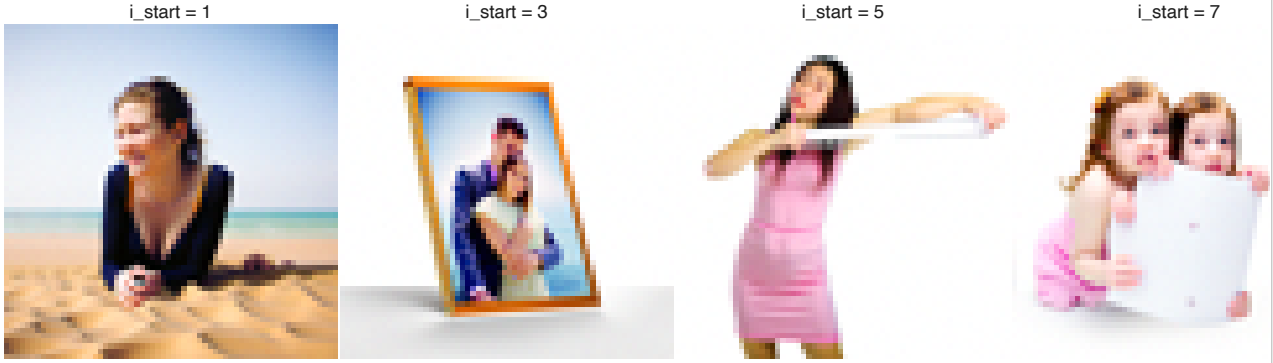
Hand Drawn Images

```
# @title Hand Drawn Images

############
### N.B. ###
############

# This code is from: https://gist.github.com/karim23657/5ad5e067c1684dbc76c93bd88bf6fa53
# The board is hardcoded to be size 300 x 600, but we're using square images
# so please just draw in the center of the board
# In addition, this gist may be taken down or change location,
# which will break things in the future

############
############

from base64 import b64decode
from IPython.display import HTML
from google.colab.output import eval_js
import urllib.request
board_html = urllib.request.urlopen('https://gist.githubusercontent.com/karim23657/5ad5e067c1684dbc76c93bd88bf6f

def draw(filename='drawing.png'):
  display(HTML(board_html))
  data = eval_js('triggerImageToServer')
  binary = b64decode(data.split(',')[1])
  with open(filename, 'wb') as f:
    f.write(binary)
  return Image.open(filename).convert('RGB')

drawn_im = draw('myImage.png').resize((64,64))
drawn_im = process_pil_im(drawn_im)
```



```
prompt_embeds = prompt_embeds_dict["a high quality photo"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

imgs_dict = []
for i_start in tqdm([1, 3, 5, 7, 10, 20]):
  # Add noise
  t = strided_timesteps[i_start]
  im_noisy = forward(drawn_im, t).half().to(device)

  # Denoise
  clean = iterative_denoise_cfg(im_noisy,
                                i_start=i_start,
                                prompt_embeds=prompt_embeds,
                                uncond_prompt_embeds=uncond_prompt_embeds,
                                timesteps=strided_timesteps,
                                display=False)

  # Add to dict to display later
  imgs_dict.append(clean)

imgs_dict.append(drawn_im[0].permute(1, 2, 0)/2 +0.5)
titles = [f"i_start = {i}" for i in [1, 3, 5, 7, 10, 20]] +[f"original"]
media.show_images(imgs_dict, titles = titles, height = 250)
```

```
100%|██████████| 6/6 [00:23<00:00,  3.96s/it]
```




```
fig, axs = plt.subplots(1, len(imgs_dict), figsize=(4 * len(imgs_list), 4))

for i, (img, title) in enumerate(zip(imgs_dict, titles)):
    img = to_img(img)      # already defined in your notebook
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```


```
fig, axs = plt.subplots(1, len(imgs_dict), figsize=(4 * len(imgs_list), 4))

for i, (img, title) in enumerate(zip(imgs_dict, titles)):
    img = to_img(img)      # already defined in your notebook
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

1.7.2 Inpainting

We can use the same procedure to implement inpainting (following the RePaint paper). That is, given an image x_{orig}, and a binary mask $\mathbf{m}$, we can create a new image that has the same content where $\mathbf{m}$ is 0, but new content wherever $\mathbf{m}$ is 1.

To do this, we can run the diffusion denoising loop. But at every step, after obtaining x_t, we "force" x_t to have the same pixels as x_{orig} where $\mathbf{m}$ is 0, i.e.:

$$x_t \leftarrow \mathbf{m}x_t + (1 - \mathbf{m})\text{forward}(x_{orig}, t) \tag{5}$$

Essentially, we leave everything inside the edit mask alone, but we replace everything outside the edit mask with our original image -- with the correct amount of noise added for timestep t.

Please implement this below, and edit the picture to inpaint the top of the Campanile.

Deliverables

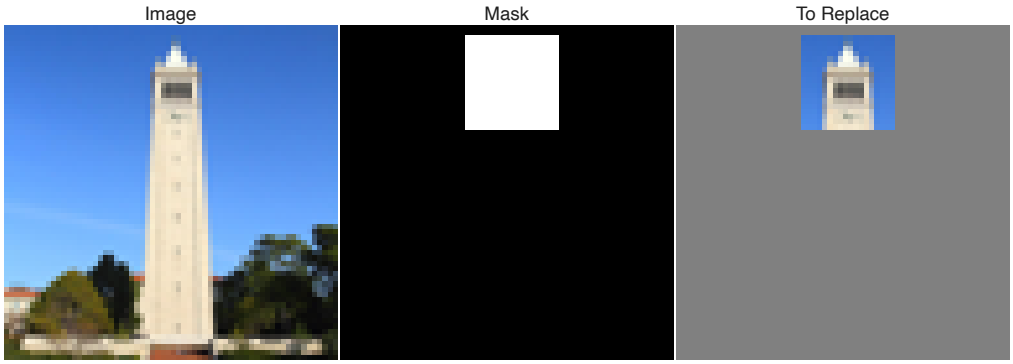
- A properly implemented `inpaint` function
- The Campanile inpainted (feel free to use your own mask)
- 2 of your own images edited (come up with your own mask)
    - look at the results from this paper for inspiration

Hints

- Reuse the `forward` function you implemented earlier
- Because we are using the diffusion model for tasks it was not trained for, you may have to run the sampling process a few times before you get a nice result.
- You can copy and paste from your `iterative_denoise_cfg` function. To get inpainting to work should only require (roughly) 1-2 additional lines and a few small changes.

Run the following cell to get the `mask` for inpainting:

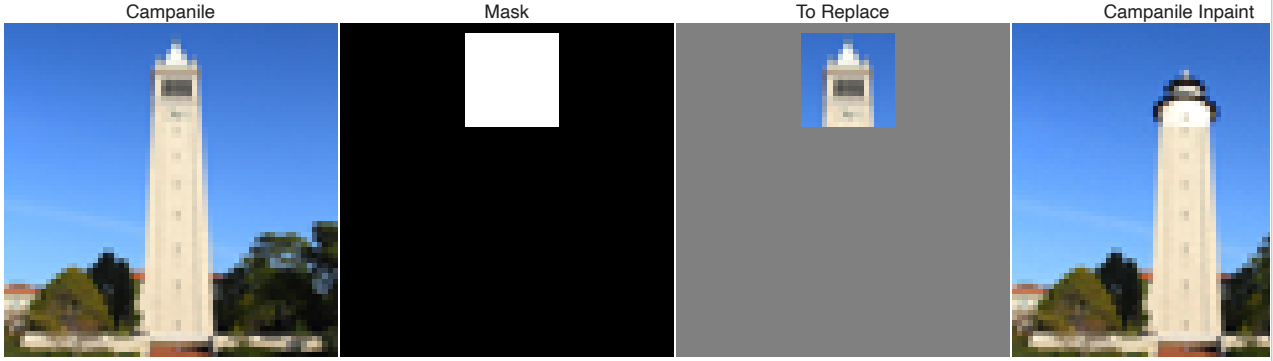
```
# EDIT ME!
# Make mask
mask = torch.zeros_like(test_im)
mask[:, :, 2:20, 24:42] = 1.0
mask = mask.to(device)

# Visualize mask
media.show_images({
    'Image': test_im[0].permute(1,2,0) / 2. + 0.5,
    'Mask': mask.cpu()[0].permute(1,2,0),
    'To Replace': (test_im * mask.cpu())[0].permute(1,2,0) / 2. + 0.5,
}, height = 250)
```



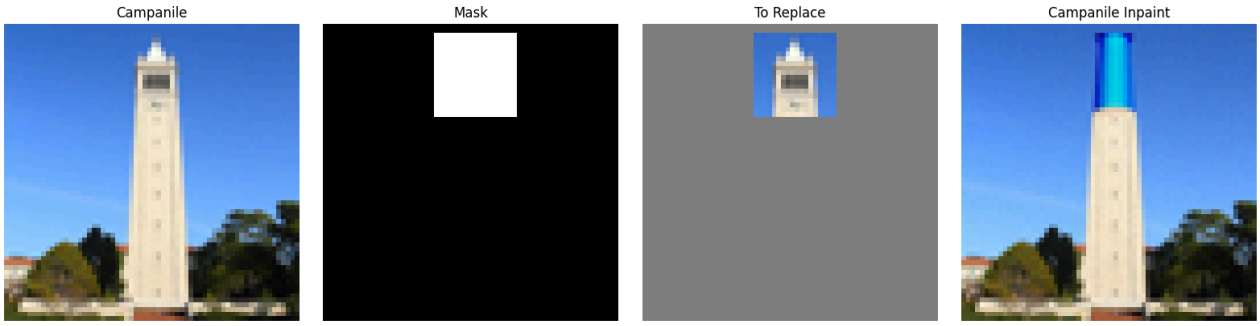

```
Start coding or generate with AI.
```

```
def inpaint(original_image, mask, prompt_embeds, uncond_prompt_embeds, timesteps, scale=7):
    image = original_image.half().to(device)

    with torch.no_grad():
        for i in range(len(timesteps) - 1):


            t = timesteps[i]
            prev_t = timesteps[i + 1]

            alpha_cumprod = alphas_cumprod[t]
            alpha_cumprod_prev = alphas_cumprod[prev_t]
            alpha = alpha_cumprod / alpha_cumprod_prev
            beta = 1 - alpha

            model_output = stage_1.unet(
                image,
                t,
                encoder_hidden_states=prompt_embeds,
                return_dict=False
            )[0]

            uncond_model_output = stage_1.unet(
                image,
                t,
                encoder_hidden_states=uncond_prompt_embeds,
                return_dict=False
            )[0]

            # Split into noise + variance
            noise_c, predicted_variance = torch.split(model_output, image.shape[1], dim=1)
            noise_u, _ = torch.split(uncond_model_output, image.shape[1], dim=1)

            noise_est = noise_u + scale * (noise_c - noise_u)
            x0_est = (image - torch.sqrt(1 - alpha_cumprod) * noise_est) / torch.sqrt(alpha_cumprod)
            pred_prev_image = (
                (torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cumprod) * x0_est +
                (torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - alpha_cumprod) * image
            )
            pred_prev_image = add_variance(predicted_variance, t, pred_prev_image)
            # ==== end of code ====
            noised_original = forward(original_image, prev_t).half().to(device)
            pred_prev_image = mask * pred_prev_image + (1 - mask) * noised_original

            # Step forward
            image = pred_prev_image

        clean = (image[0].permute(1, 2, 0)/2 +0.5).cpu().detach().numpy()

    return clean
```

```
inpainted = inpaint(
    original_image=test_im,
    mask=mask.half(),
    prompt_embeds=prompt_embeds,
    uncond_prompt_embeds=uncond_prompt_embeds,
    timesteps=strided_timesteps,
    scale=7
)

media.show_images({
    'Campanile': test_im[0].permute(1,2,0) / 2. + 0.5,
    'Mask': mask.cpu()[0].permute(1,2,0),
    'To Replace': (test_im * mask.cpu())[0].permute(1,2,0) / 2. + 0.5,
    'Campanile Inpaint': inpainted
}, height = 250)
```

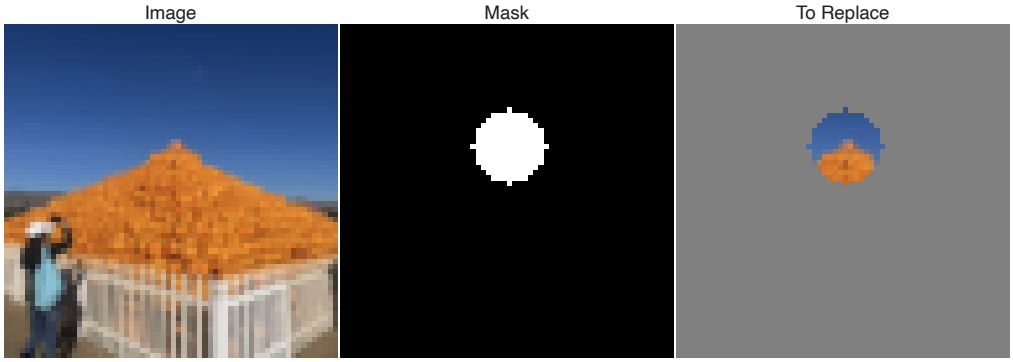


```
imgs_dict = {
    'Campanile': test_im[0].permute(1,2,0) / 2. + 0.5,
    'Mask': mask.cpu()[0].permute(1,2,0),
    'To Replace': (test_im * mask.cpu())[0].permute(1,2,0) / 2. + 0.5,
    'Campanile Inpaint': inpainted
}

fig, axs = plt.subplots(1, len(imgs_dict), figsize=(4 * len(imgs_dict), 4))

for i, (title, img) in enumerate(imgs_dict.items()):
    img = to_img(img)   # your existing conversion helper
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



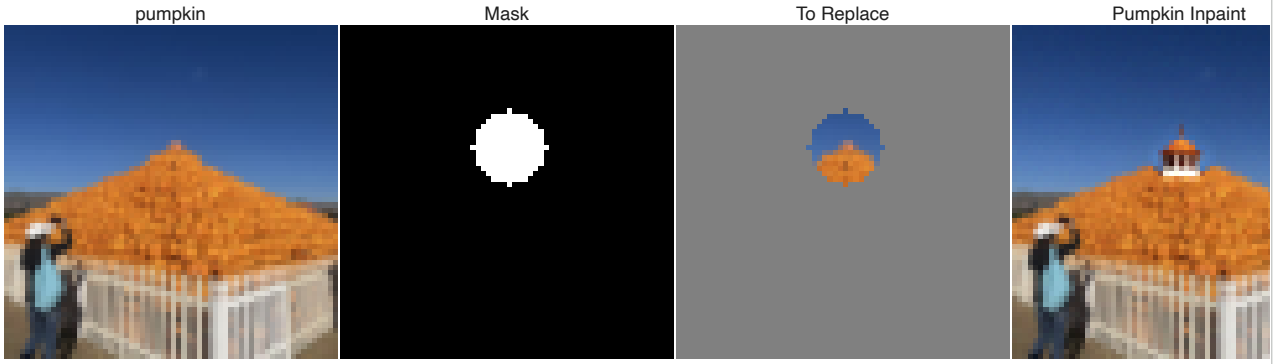

```
path = "/content/drive/MyDrive/cs180_project5_data/pumpkin_patch.jpg"
pumpkin = Image.open(path)
pumpkin = process_pil_im(pumpkin).to(device)

_, _, H, W = pumpkin.shape

# Choose circle center (y, x)
cy, cx = 23, 32   # adjust after preview

radius = 7

# Make coordinate grid
Y, X = torch.meshgrid(torch.arange(H), torch.arange(W), indexing='ij')
dist = torch.sqrt((X - cx)**2 + (Y - cy)**2)

# Create mask: 1 inside circle, 0 outside
mask_circle = (dist <= radius).float()[None, None, :, :]

mask_circle = mask_circle.to(device).half()

# Display preview
media.show_images({
    "Image": (pumpkin[0].permute(1,2,0) / 2 + 0.5).cpu(),
    "Mask": mask_circle[0].permute(1,2,0).cpu(),
    "To Replace": ((pumpkin * mask_circle)[0].permute(1,2,0) / 2 + 0.5).cpu(),
}, height = 250
                  )
```

```
Processed image
```

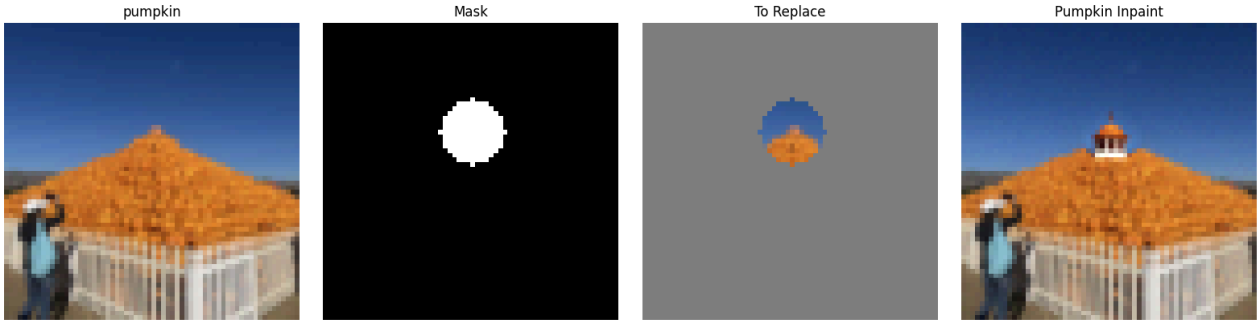



```
print
```

```
inpainted_pumpkin = inpaint(
    original_image=pumpkin,
    mask=mask_circle,
    prompt_embeds=prompt_embeds,
    uncond_prompt_embeds=uncond_prompt_embeds,
    timesteps=strided_timesteps,
    scale=7
)

media.show_images({
    'pumpkin': (pumpkin[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Mask': mask_circle.cpu()[0].permute(1,2,0),
    "To Replace": ((pumpkin * mask_circle)[0].permute(1,2,0) / 2 + 0.5).cpu(),
    'Pumpkin Inpaint': inpainted_pumpkin
}, height = 250)
```



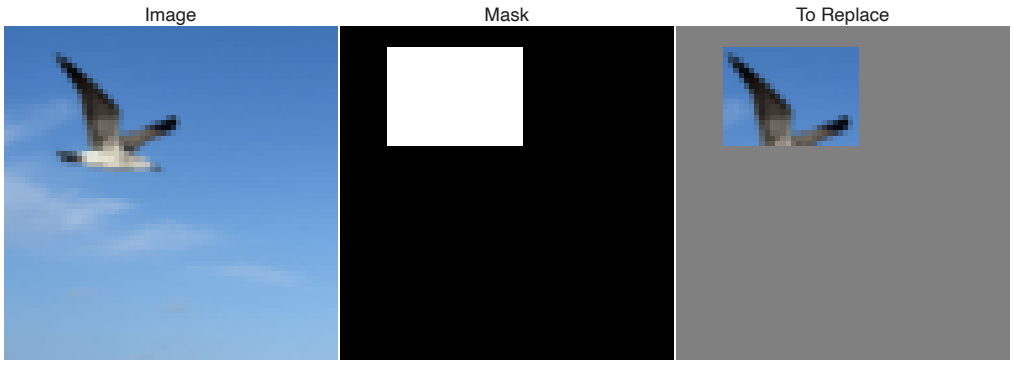

```
imgs_dict = {
    'pumpkin': (pumpkin[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Mask': mask_circle.cpu()[0].permute(1,2,0),
    'To Replace': ((pumpkin * mask_circle)[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Pumpkin Inpaint': inpainted_pumpkin
}

fig, axs = plt.subplots(1, len(imgs_dict), figsize=(4 * len(imgs_dict), 4))

for i, (title, img) in enumerate(imgs_dict.items()):
    img = to_img(img)

    # If image has 1 channel → grayscale mask
    if img.ndim == 2 or (img.ndim == 3 and img.shape[-1] == 1):
        axs[i].imshow(img.squeeze(), cmap="gray")
    else:
        axs[i].imshow(img)

    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



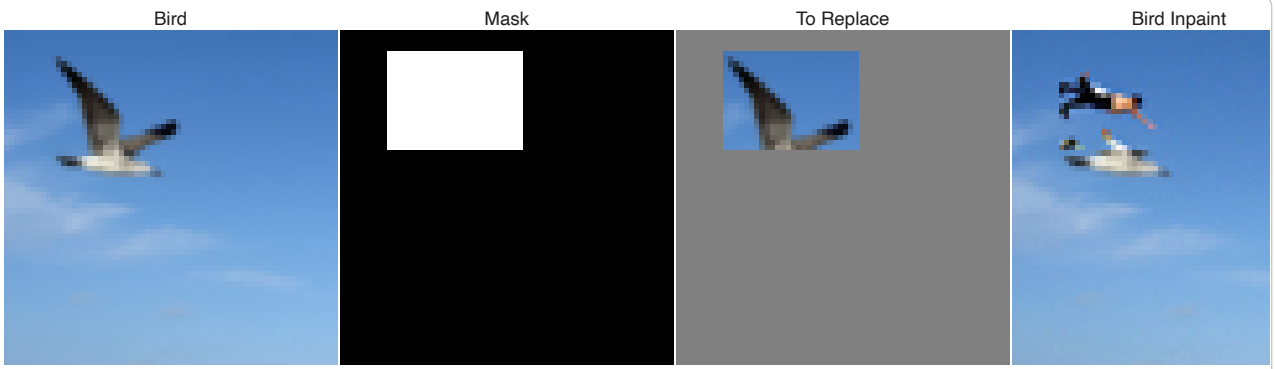

```
path = "/content/drive/MyDrive/cs180_project5_data/bird.jpg"
bird = Image.open(path)
bird = process_pil_im(bird).to(device)
```

```
Processed image
```

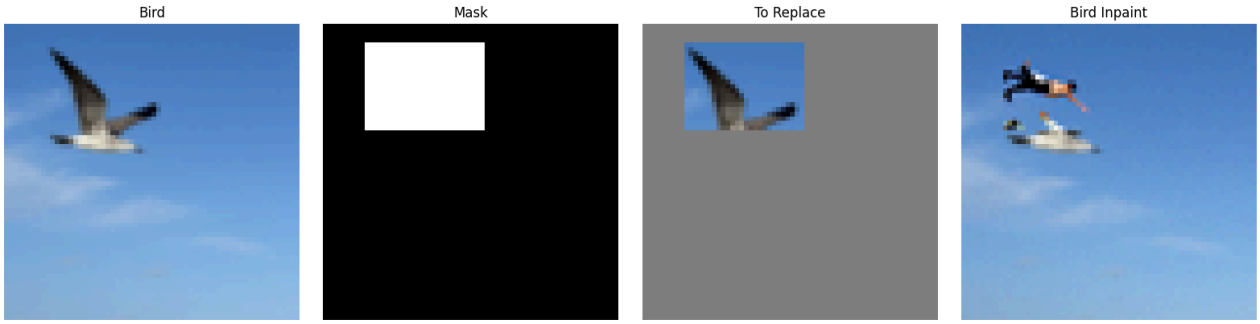
```
mask_bird = torch.zeros_like(test_im)
mask_bird[:, :, 4:23, 9:35] = 1.0
mask_bird = mask_bird.half().to(device)

# Visualize mask
media.show_images({
    "Image": (bird[0].permute(1,2,0) / 2 + 0.5).cpu(),
    "Mask": mask_bird[0].permute(1,2,0).cpu(),
    "To Replace": ((bird * mask_bird)[0].permute(1,2,0) / 2 + 0.5).cpu(),
}, height = 250)
```




```
inpainted_bird = inpaint(
    original_image=bird,
    mask=mask_bird,
    prompt_embeds=prompt_embeds,
    uncond_prompt_embeds=uncond_prompt_embeds,
    timesteps=strided_timesteps,
    scale=7
)

media.show_images({
    'Bird': (bird[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Mask': mask_bird.cpu()[0].permute(1,2,0),
    "To Replace": ((bird * mask_bird)[0].permute(1,2,0) / 2 + 0.5).cpu(),
    'Bird Inpaint': inpainted_bird
}, height = 250)
```




```
imgs_dict = {
    'Bird': (bird[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Mask': mask_bird.cpu()[0].permute(1,2,0),
    'To Replace': ((bird * mask_bird)[0].permute(1,2,0) / 2. + 0.5).cpu(),
    'Bird Inpaint': inpainted_bird
}

fig, axs = plt.subplots(1, len(imgs_dict), figsize=(4 * len(imgs_dict), 4))

for i, (title, img) in enumerate(imgs_dict.items()):
    img = to_img(img)     # your existing safe converter

    # 💡 Display mask in grayscale, NOT viridis (avoids purple)
    if img.ndim == 2 or (img.ndim == 3 and img.shape[-1] == 1):
        axs[i].imshow(img.squeeze(), cmap='gray')
    else:
        axs[i].imshow(img)

    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



1.7.3 Text-Conditioned Image-to-image Translation

Now, we will do the same thing as the previous section, but guide the projection with a text prompt. This is no longer pure "projection to the natural image manifold" but also adds control using language. This is simply a matter of changing the prompt from `"a high quality photo"` to any of your prompt!

Deliverables

- Edits of the Campanile, using the given prompt at noise levels [1, 3, 5, 7, 10, 20]
- Edits of 2 of your own test images, using the same procedure.

Hints

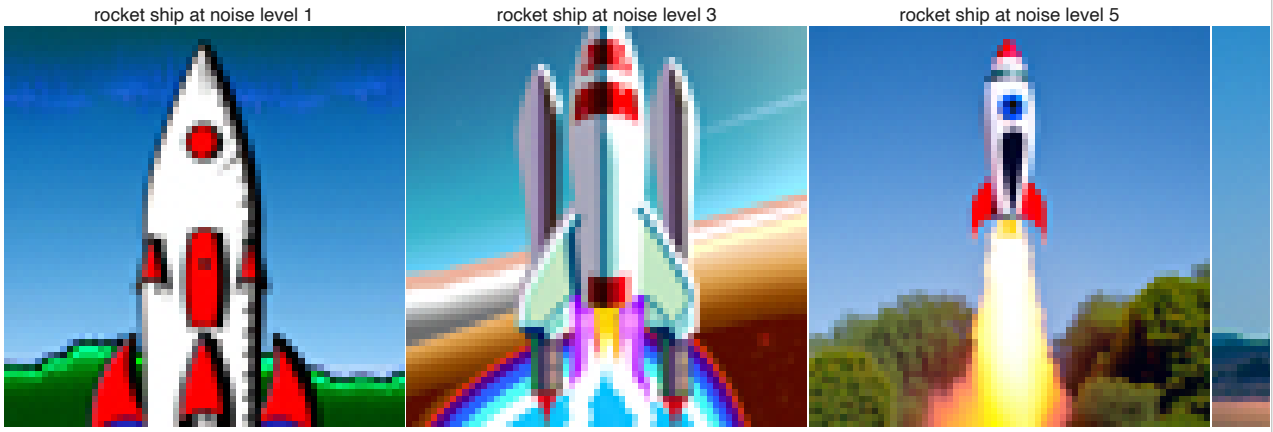
- The images should gradually look more like original image, but also look like the text prompt.

```
prompt_embeds = prompt_embeds_dict["a rocket ship"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)
```

```
noise_levels = [1, 3, 5, 7, 10, 20]

edited_outputs = []

for i_start in noise_levels:
    print(f"Running edit with i_start = {i_start} ...")

    # Add noise
    t = strided_timesteps[i_start]
    im_noisy = forward(test_im, t).half().to(device)

    # Denoise with CFG
    edited = iterative_denoise_cfg(
        im_noisy,
        i_start=i_start,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edited_outputs.append(edited)
```

```
Running edit with i_start = 1 ...
Running edit with i_start = 3 ...
Running edit with i_start = 5 ...
Running edit with i_start = 7 ...
Running edit with i_start = 10 ...
Running edit with i_start = 20 ...
```

```
titles = [f"rocket ship at noise level {i}" for i in noise_levels] + [f'campanile']
edited_outputs.append(test_im[0].permute(1, 2, 0)/2 + 0.5)
media.show_images(edited_outputs, titles=titles, height=300)
```

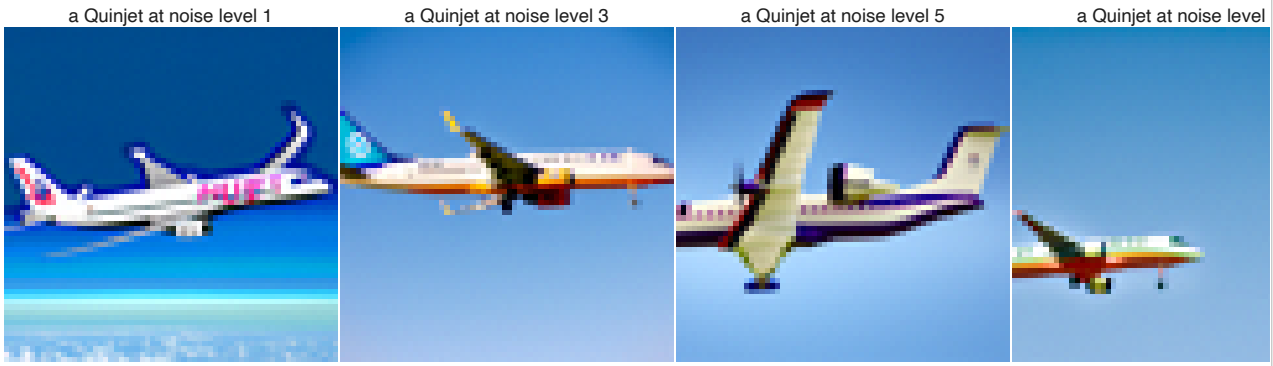



```
fig, axs = plt.subplots(1, len(edited_outputs), figsize=(4 * len(edited_outputs), 4))

for i, (img, title) in enumerate(zip(edited_outputs, titles)):
    img = to_img(img)    # your existing converter

    # handle grayscale masks automatically
    if img.ndim == 2 or (img.ndim == 3 and img.shape[-1] == 1):
        axs[i].imshow(img.squeeze(), cmap='gray')
    else:
        axs[i].imshow(img)

    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```


```
prompt_embeds = prompt_embeds_dict["a Quinjet"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

noise_levels = [1, 3, 5, 7, 10, 20]

edited_outputs2 = []

for i_start in noise_levels:
    print(f"Running edit with i_start = {i_start} ...")

    # Add noise
    t = strided_timesteps[i_start]
    im_noisy = forward(bird, t).half().to(device)

    # Denoise with CFG
    edited = iterative_denoise_cfg(
        im_noisy,
        i_start=i_start,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edited_outputs2.append(edited)
```

```
Running edit with i_start = 1 ...
Running edit with i_start = 3 ...
Running edit with i_start = 5 ...
Running edit with i_start = 7 ...
Running edit with i_start = 10 ...
Running edit with i_start = 20 ...
```

```
titles = [f"a Quinjet at noise level {i}" for i in noise_levels] + [f'bird']
edited_outputs2.append((bird[0].permute(1, 2, 0)/2 + 0.5).cpu())
media.show_images(edited_outputs2, titles, height = 250)
```



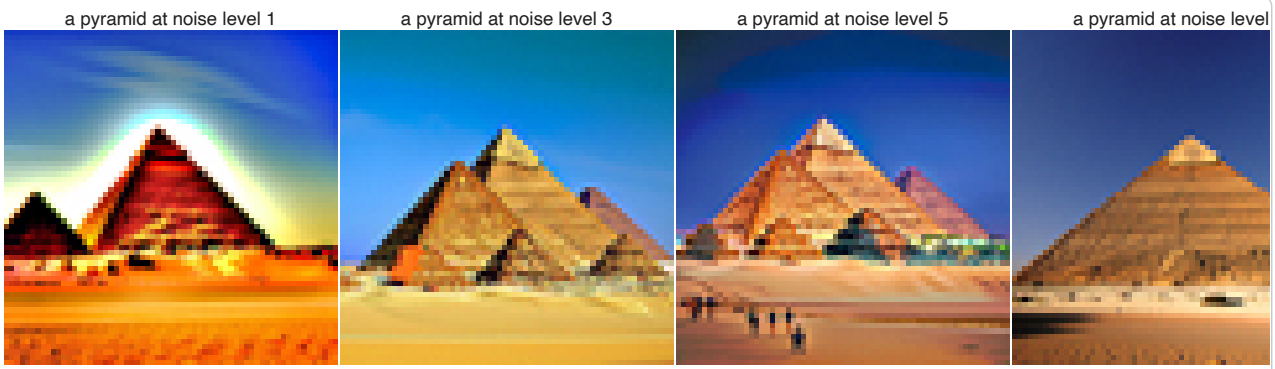

```
fig, axs = plt.subplots(1, len(edited_outputs2), figsize=(4 * len(edited_outputs2), 4))
```

```
for i, (img, title) in enumerate(zip(edited_outputs2, titles)):
    img = to_img(img)   # your already-defined safe converter

    # If grayscale → display properly, avoid purple
    if img.ndim == 2 or (img.ndim == 3 and img.shape[-1] == 1):
        axs[i].imshow(img.squeeze(), cmap='gray')
    else:
        axs[i].imshow(img)

    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```

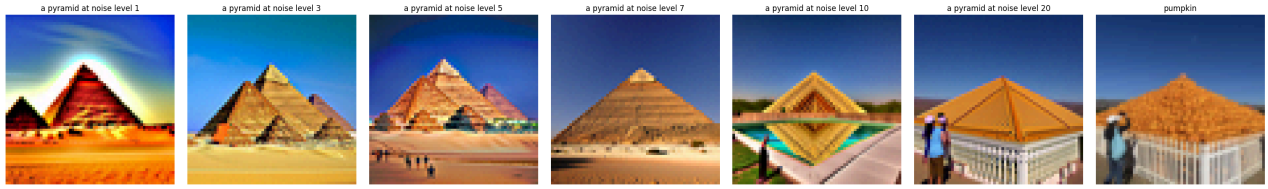

```
prompt_embeds = prompt_embeds_dict["a pyramid"].half().to(device)
uncond_prompt_embeds = prompt_embeds_dict[''].half().to(device)

noise_levels = [1, 3, 5, 7, 10, 20]

edited_outputs3 = []

for i_start in noise_levels:
    print(f"Running edit with i_start = {i_start} ...")

    # Add noise
    t = strided_timesteps[i_start]
    im_noisy = forward(pumpkin, t).half().to(device)

    # Denoise with CFG
    edited = iterative_denoise_cfg(
        im_noisy,
        i_start=i_start,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        scale=7,
        display=False
    )

    edited_outputs3.append(edited)
```

```
Running edit with i_start = 1 ...
Running edit with i_start = 3 ...
Running edit with i_start = 5 ...
Running edit with i_start = 7 ...
Running edit with i_start = 10 ...
Running edit with i_start = 20 ...
```

```
titles = [f"a pyramid at noise level {i}" for i in noise_levels] + [f'pumpkin']
edited_outputs3.append((pumpkin[0].permute(1, 2, 0)/2 + 0.5).cpu())
media.show_images(edited_outputs3, titles, height = 250)
```




```
fig, axs = plt.subplots(1, len(edited_outputs3), figsize=(4 * len(edited_outputs3), 4))

for i, (img, title) in enumerate(zip(edited_outputs3, titles)):
    img = to_img(img)   # your existing safe converter

    # Show grayscale masks correctly (avoid purple)
    if img.ndim == 2 or (img.ndim == 3 and img.shape[-1] == 1):
        axs[i].imshow(img.squeeze(), cmap='gray')
    else:
        axs[i].imshow(img)

    axs[i].set_title(title)
    axs[i].axis("off")

plt.tight_layout()
plt.show()
```



1.8 Visual Anagrams

In this part, we are finally ready to implement Visual Anagrams and create optical illusions with diffusion models. In this part, we will create an image that looks like `"an oil painting of an old man"`, but when flipped upside down will reveal `"an oil painting of people around a campfire"`.

To do this, we will denoise an image x_t at step t normally with the prompt p_1, to obtain noise estimate ϵ_1. But at the same time, we will flip x_t upside down, and denoise with the prompt p_2, to get noise estimate ϵ_2. We can flip ϵ_2 back, and average the two noise estimates. We can then perform a reverse diffusion step with the averaged noise estimate.

The full algorithm is:

$$\epsilon_1 = \text{CFG of UNet}(x_t, t, p_1)$$

$$\epsilon_2 = \text{flip}(\text{CFG of UNet}(\text{flip}(x_t), t, p_2))$$

$$\epsilon = (\epsilon_1 + \epsilon_2)/2$$

where UNet is the diffusion model UNet from before, $\text{flip}(\cdot)$ is a function that flips the image, and p_1 and p_2 are two different text prompt embeddings. And our final noise estimate is ϵ. Please implement the above algorithm and show example of an illusion.

Deliverables

- Correctly implemented `visual_anagrams` function
- 2 illusions of your choice that change appearance when you flip it upside down

Hints

- You may have to run multiple times to get a really good result for the same reasons as above

```
def make_flip_illusion(image, i_start, prompt1_embeds, prompt2_embeds,
                       uncond_prompt_embeds, timesteps, scale=7, display=True):
```

```
    x = image.clone()

    with torch.no_grad():
        for i in range(i_start, len(timesteps) - 1):

            t = timesteps[i]
            prev_t = timesteps[i + 1]

            alpha_cumprod = alphas_cumprod[t]
            alpha_cumprod_prev = alphas_cumprod[prev_t]
            alpha = alpha_cumprod / alpha_cumprod_prev
            beta = 1 - alpha

            model_out1 = stage_1.unet(
                x,
                t,
                encoder_hidden_states=prompt1_embeds,
                return_dict=False
            )[0]

            uncond_out1 = stage_1.unet(
                x,
                t,
                encoder_hidden_states=uncond_prompt_embeds,
                return_dict=False
            )[0]

            noise1, var1 = torch.split(model_out1, x.shape[1], dim=1)
            uncond_noise1, _ = torch.split(uncond_out1, x.shape[1], dim=1)

            eps1 = uncond_noise1 + scale * (noise1 - uncond_noise1)

            x_flipped = torch.flip(x, dims=[2])  # flip vertically

            model_out2 = stage_1.unet(
                x_flipped,
                t,
                encoder_hidden_states=prompt2_embeds,
                return_dict=False
            )[0]

            uncond_out2 = stage_1.unet(
                x_flipped,
                t,
                encoder_hidden_states=uncond_prompt_embeds,
                return_dict=False
            )[0]

            noise2, var2 = torch.split(model_out2, x.shape[1], dim=1)
            uncond_noise2, _ = torch.split(uncond_out2, x.shape[1], dim=1)

            eps1 = uncond_noise1 + scale * (noise1 - uncond_noise1)
            eps2 = uncond_noise2 + scale * (noise2 - uncond_noise2)
            eps2 = torch.flip(eps2, dims=[2])
            eps = (eps1 + eps2) / 2.0

            x0_est = (x - torch.sqrt(1 - alpha_cumprod) * eps) / torch.sqrt(alpha_cumprod)

            x_prev = (
                (torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cumprod) * x0_est
                + (torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - alpha_cumprod) * x
            )

            x_prev = add_variance(var1, t, x_prev)

            x = x_prev

    clean = (x[0].permute(1, 2, 0) / 2 + 0.5).cpu().detach().numpy()

    return clean
```

```
illusion = make_flip_illusion(
    image=torch.randn(1,3,64,64).half().to(device),
    i_start=0,
    prompt1_embeds=prompt_embeds_dict["an oil painting of an old man"].half().to(device),
    prompt2_embeds=prompt_embeds_dict["an oil painting of people around a campfire"].half().to(device),
    uncond_prompt_embeds=prompt_embeds_dict[""].half().to(device),
    timesteps=strided_timesteps,
    scale=7
)
```

1.9 Hybrid Images

In this part we'll implement Factorized Diffusion and create hybrid images just like in project 2.

In order to create hybrid images with a diffusion model we can use a similar technique as above. We will create a composite noise estimate ϵ, by estimating the noise with two different text prompts, and then combining low frequencies from one noise estimate with high frequencies of the other. The algorithm is:

$\epsilon_1 = \text{CFG of UNet}(x_t, t, p_1)$

$\epsilon_2 = \text{CFG of UNet}(x_t, t, p_2)$

$\epsilon = f_{\text{lowpass}}(\epsilon_1) + f_{\text{highpass}}(\epsilon_2)$

where UNet is the diffusion model UNet, f_{lowpass} is a low pass function, f_{highpass} is a high pass function, and p_1 and p_2 are two different text prompt embeddings. Our final noise estimate is ϵ. Please show an example of a hybrid image using this technique (you may have to run multiple times to get a really good result for the same reasons as above). We recommend that you use a gaussian blur of kernel size 33 and sigma 2.

Deliverables