CS5670 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. 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.

```
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
```

```
76.1/76.1 MB 32.9 MB/s eta 0:00:0
363.4/363.4 MB 4.1 MB/s eta 0:00:
13.8/13.8 MB 124.6 MB/s eta 0:00:
24.6/24.6 MB 100.6 MB/s eta 0:00:
883.7/883.7 kB 53.5 MB/s eta 0:00
664.8/664.8 MB 1.7 MB/s eta 0:00:
211.5/211.5 MB 11.7 MB/s eta 0:00
56.3/56.3 MB 43.1 MB/s eta 0:00:0
127.9/127.9 MB 18.9 MB/s eta 0:00
207.5/207.5 MB 3.3 MB/s eta 0:00:
21.1/21.1 MB 112.1 MB/s eta 0:00:
1.6/1.6 MB 77.1 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'
```

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/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: Use
The secret `HF_TOKEN` does not exist in your Colab secrets.
```

```
To authenticate with the Hugging Face Hub, create a token in your settings tar
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access pu
  warnings.warn(
model_index.json: 100%                                604/604 [00:00<00:00, 65.6kB/s]

A mixture of fp16 and non-fp16 filenames will be loaded.
Loaded fp16 filenames:
[unet/diffusion_pytorch_model.fp16.safetensors, safety_checker/model.fp16.safe
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 [00:08<00:00,  1.16it/s]

preprocessor_config.json: 100%                        518/518 [00:00<00:00, 18.9kB/s]

config.json: 100%                                     1.63k/1.63k [00:00<00:00, 51.4kB/s]

special_tokens_map.json: 100%                         2.20k/2.20k [00:00<00:00, 129kB/s]

tokenizer_config.json: 100%                           2.54k/2.54k [00:00<00:00, 65.1kB/s]

config.json: 100%                                     4.57k/4.57k [00:00<00:00, 130kB/s]

model.fp16.safetensors: 100%                          608M/608M [00:02<00:00, 245MB/s]

spiece.model: 100%                                    792k/792k [00:00<00:00, 10.0MB/s]

diffusion_pytorch_model.fp16.safetensors: 100%        1.87G/1.87G [00:08<00:00, 254MB/s]

diffusion_pytorch_model.safetensors: 100%             15.5k/15.5k [00:00<00:00, 1.38MB/s]

config.json: 100%                                     74.0/74.0 [00:00<00:00, 7.97kB/s]

scheduler_config.json: 100%                           454/454 [00:00<00:00, 42.8kB/s]

Loading pipeline components...: 100%                  6/6 [00:01<00:00,  3.69it/s]
You are using the default legacy behaviour of the <class 'transformers.models.
model_index.json: 100%                                692/692 [00:00<00:00, 90.4kB/s]

A mixture of fp16 and non-fp16 filenames will be loaded.
Loaded fp16 filenames:
[unet/diffusion_pytorch_model.fp16.safetensors, safety_checker/model.fp16.safe
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 [00:10<00:00,  1.20s/it]

preprocessor_config.json: 100%                        518/518 [00:00<00:00, 26.8kB/s]
```

```
scheduler_config.json: 100%                                   424/424 [00:00<00:00, 13.7kB/s]
model.fp16.safetensors: 100%                                  608M/608M [00:02<00:00, 251MB/s]
spiece.model: 100%                                            792k/792k [00:00<00:00, 6.26MB/s]
scheduler_config.json: 100%                                   454/454 [00:00<00:00, 35.3kB/s]
config.json: 100%                                             4.92k/4.92k [00:00<00:00, 294kB/s]
diffusion_pytorch_model.fp16.safetensors: 100%                2.49G/2.49G [00:10<00:00, 238MB/s]
tokenizer_config.json: 100%                                   2.50k/2.50k [00:00<00:00, 92.3kB/s]
config.json: 100%                                             1.72k/1.72k [00:00<00:00, 117kB/s]
special_tokens_map.json: 100%                                 2.20k/2.20k [00:00<00:00, 252kB/s]
diffusion_pytorch_model.safetensors: 100%                     15.5k/15.5k [00:00<00:00, 1.51MB/s]
config.json: 100%                                             74.0/74.0 [00:00<00:00, 8.32kB/s]
Loading pipeline components...: 100%                          7/7 [00:00<00:00, 12.30it/s]
IFSuperResolutionPipeline {
  "_class_name": "IFSuperResolutionPipeline",
  "_diffusers_version": "0.32.2",
  "_name_or_path": "DeepFloyd/IF-II-L-v1.0",
  "feature_extractor": [
    "transformers",
    "CLIPImageProcessor"
  ],
  "image_noising_scheduler": [
    "diffusers",
    "DDPMScheduler"
  ],
  "requires_safety_checker": true,
  "safety_checker": [
    "deepfloyd_if",
    "IFSafetyChecker"
  ],
  "scheduler": [
    "diffusers",
    "DDPMScheduler"
  ],
  "text_encoder": [
    null,
    null
  ],
  "tokenizer": [
    "transformers",
    ],
    "unet": [
      "diffusers",
      "UNet2DConditionModel"
    ],
    "watermarker": [
      "deepfloyd_if",
      "IFWatermarker"
    ]
  }
```

Disclaimer about Text Embeddings

DeepFloyd was trained as a text-to-image model, which takes text prompts as input and outputs images that are aligned with the text. Throughout this notebook, you will see that we ask you to generate with the prompt "a high quality photo". We want you to think of this as a "null" prompt that doesn't have any specific meaning, and is simply a way for the model to do unconditional generation. You can view this as using the diffusion model to "force" a noisy image onto the "manifold" of real images.

In the later sections, we will guide this project with a more detailed text prompt.

Downloading Precomputed Text Embeddings

Because the text encoder is very large, and barely fits on a free tier Colab GPU, we have precomputed a couple of text embeddings for you to try. This should hopefully save some headaches from GPU out of memory errors. At the end of the homework, we provide you code if you want to try your own text prompts. If you'd like, you can pay $10 for Colab Pro and avoid needing to load the two models on different sessions.

```
!wget https://www.cs.cornell.edu/courses/cs5670/2025sp/projects/5_project/files/p

prompt_embeds_dict = torch.load('prompt_embeds_dict.pth')

pprint(list(prompt_embeds_dict.keys()))
```

```
--2025-04-18 17:08:22--  https://www.cs.cornell.edu/courses/cs5670/2025sp/pro
Resolving www.cs.cornell.edu (www.cs.cornell.edu)... 132.236.207.53
Connecting to www.cs.cornell.edu (www.cs.cornell.edu)|132.236.207.53|:443...
HTTP request sent, awaiting response... 200 OK
Length: 8836210 (8.4M)
Saving to: ‘prompt_embeds_dict.pth’

prompt_embeds_dict. 100%[===================>]   8.43M  23.3MB/s    in 0.4s

2025-04-18 17:08:23 (23.3 MB/s) - ‘prompt_embeds_dict.pth’ saved [8836210/883

['an oil painting of a snowy mountain village',
 'a photo of the amalfi cost',
 'a photo of a man',
 'a photo of a hipster barista',
 'a photo of a dog',
 'an oil painting of people around a campfire',
 'an oil painting of an old man',
 'a lithograph of waterfalls',
 'a lithograph of a skull',
 'a man wearing a hat',
 'a high quality photo',
 '',
 'a rocket ship',
 'a pencil']
```

Seed your Work

To reproduce your code, please use a random seed from this point onward.

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

YOUR_SEED = 17102002
seed_everything(YOUR_SEED)
```

Sampling from the Model [TODO 0]

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`.

Delivarables

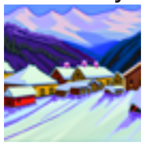
- For the 3 text prompts that we provide, display the caption and the output of the model. Briefly reflect on the quality of the outputs and their relationships to the text prompts.
- Try a different `num_inference_steps` for at least one of the images, showing before and after. Make sure to try at least 2 different `num_inference_steps` values.
- Report the random seed you used. You should use the same seed for all subsequent parts.

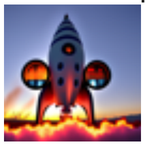
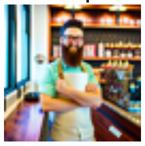
```
# Get prompt embeddings from the precomputed cache.
# `prompt_embeds` is of shape [N, 77, 4096]
# 77 comes from the max sequence length that deepfloyd will take
# and 4096 comes from the embedding dimension of the text encoder
# `negative_prompt_embeds` is the same shape as `prompt_embeds` and is 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 = [
    'an oil painting of a snowy mountain village',
    'a man wearing a hat',
    "a rocket ship",    "a photo of a hipster barista"

]
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=20,
    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=20,
    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]
print("Stage 1 Outputs (64x64) with inference steps 20")
media.show_images(
    stage_1_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts)
print("Stage 2 Outputs (256x256) with inference steps 20")
media.show_images(
    stage_2_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts)
```

```
100%                                                  20/20 [00:02<00:00, 13.24it/s]

100%                                                  20/20 [00:04<00:00,  4.14it/s]
Stage 1 Outputs (64x64) with inference steps 20
```

an oil painting of a snowy mountain village a man wearing a hat a rocket ship a photo of a hipster barista



```
Stage 2 Outputs (256x256) with inference steps 20
```

an oil painting of a snowy mountain village

a man wearing a hat

a rocke

Write you thoughts here

Reflections on the initial 3 outputs:

`The image is cleared and more detailed in stage 2 compared to stage 1. The image matches the prompt accurately`

Random seed used:

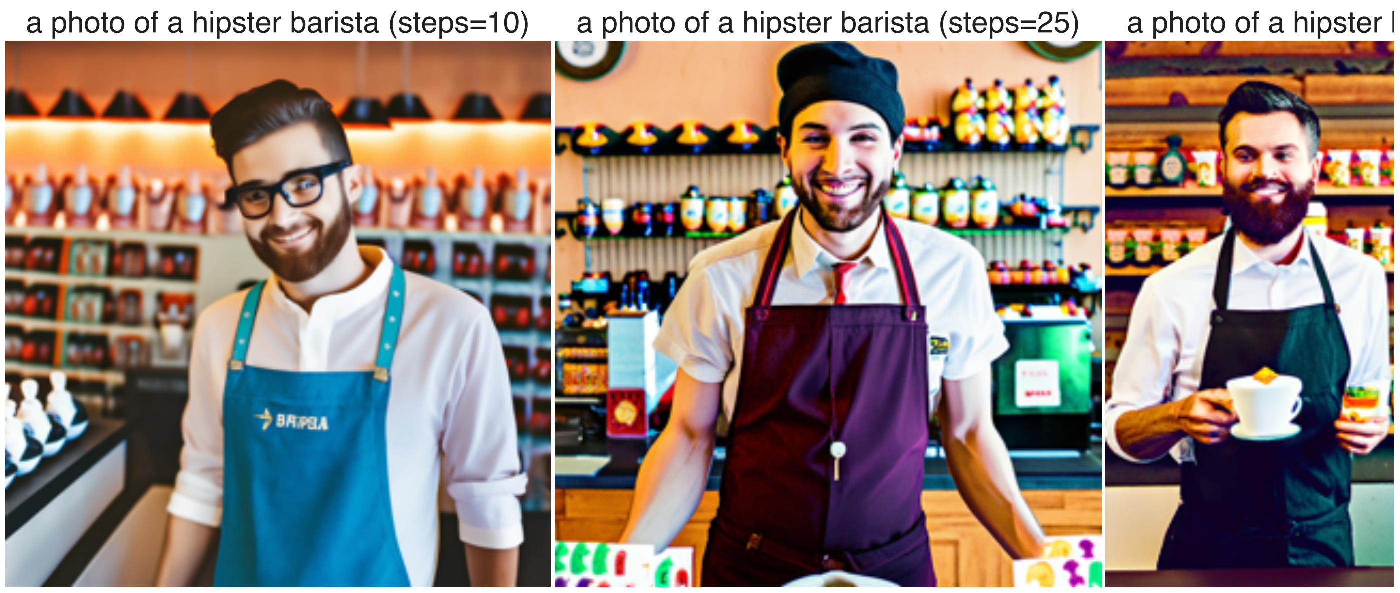
`17102002`

Try different num_inference_steps for one prompt and compare results

- Prompt: "a photo of a hipster barista"
- Steps tried: 10,25,60, 100
- What changed in quality? At steps 20 the images were generated quickly but are smooth and lack detail, when the steps were increased to 100 it took much longer to generate but the images were detailed and sharper.

```
# Try different num_inference_steps for one prompt and compare results
prompt_index = 3  # 0 = snowy mountain, 1 = man with hat, 2 = rocket ship
single_prompt = prompts[prompt_index]
prompt_embed = prompt_embeds[prompt_index:prompt_index+1]
negative_embed = negative_prompt_embeds[prompt_index:prompt_index+1]

# Try two different inference step values
steps_list = [10,25,60, 100]
stage2_outputs = []

for steps in steps_list:
    print(f"Sampling with num_inference_steps = {steps}")

    # Stage 1
    stage1_out = stage_1(
        prompt_embeds=prompt_embed,
        negative_prompt_embeds=negative_embed,
        num_inference_steps=steps,
        output_type="pt"
    ).images

    # Stage 2
    stage2_out = stage_2(
        image=stage1_out,
        prompt_embeds=prompt_embed,
        negative_prompt_embeds=negative_embed,
        num_inference_steps=steps,
        output_type="pt"
    ).images

    stage2_outputs.append(stage2_out)

# Display side-by-side comparison
media.show_images(
    torch.cat(stage2_outputs).permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=[f"{single_prompt} (steps={s})" for s in steps_list]
)
```

```
Sampling with num_inference_steps = 10
100%                                                10/10 [00:00<00:00, 16.19it/s]

100%                                                10/10 [00:00<00:00, 10.38it/s]
Sampling with num_inference_steps = 25
100%                                                25/25 [00:01<00:00, 18.16it/s]

100%                                                25/25 [00:02<00:00, 10.73it/s]
Sampling with num_inference_steps = 60
100%                                                60/60 [00:03<00:00, 18.53it/s]

100%                                                60/60 [00:05<00:00, 10.80it/s]
Sampling with num_inference_steps = 100
100%                                                100/100 [00:05<00:00, 18.54it/s]

100%                                                100/100 [00:09<00:00, 11.01it/s]
```

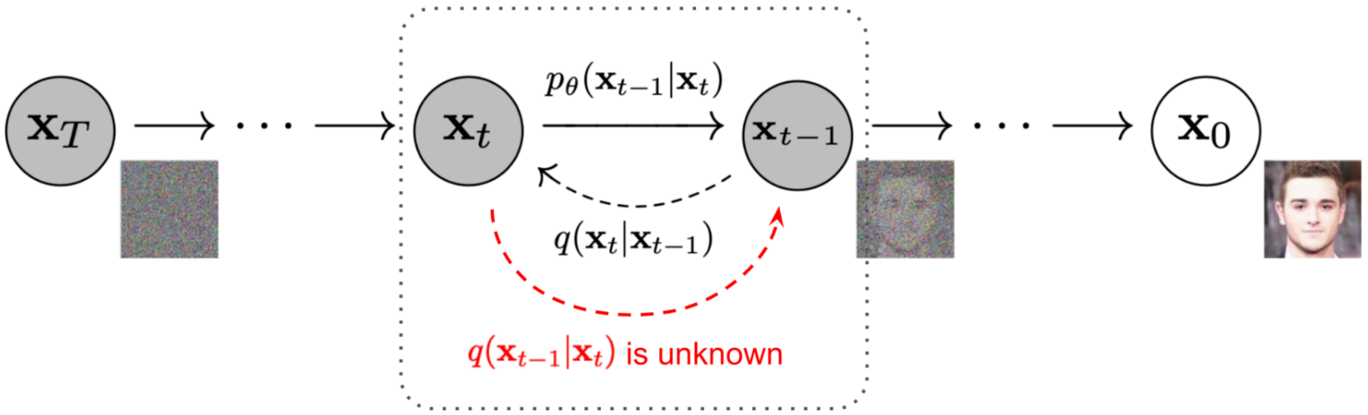

a photo of a hipster barista (steps=10)
a photo of a hipster barista (steps=25)
a photo of a hipster

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


$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is unknown
$\mathbf{x}_0$

(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 and downloads a test image that we will work with.

```
# Get scheduler parameters
alphas_cumprod = stage_1.scheduler.alphas_cumprod

# Get test image
!wget https://www.cs.cornell.edu/courses/cs5670/2025sp/projects/5_project/files/c
test_im = Image.open('cornell_tower.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('cornell_tower.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)
```

```
--2025-04-18 17:09:21--  https://www.cs.cornell.edu/courses/cs5670/2025sp/pro
Resolving www.cs.cornell.edu (www.cs.cornell.edu)... 132.236.207.53
Connecting to www.cs.cornell.edu (www.cs.cornell.edu)|132.236.207.53|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 63935 (62K) [image/jpeg]
Saving to: ‘cornell_tower.jpg’

cornell_tower.jpg   100%[===================>]  62.44K  --.-KB/s    in 0.07s

2025-04-18 17:09:22 (887 KB/s) - ‘cornell_tower.jpg’ saved [63935/63935]

Test image:
```

1.1 Implementing the forward process [TODO 1]

Disclaimer about equations: Colab 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

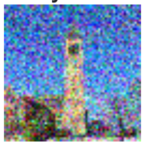
- Implement the `im_noisy = forward(im, t)` function
- Show the test image at noise level [250, 500, 750]

Hints

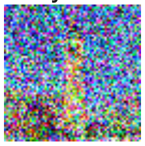
- The `torch.randn_like` function is helpful for computing ϵ.
- Use the `alphas_cumprod` variable, which contains an array of the hyperparameters, with `alphas_cumprod[t]` corresponding to $\bar{\alpha}_t$.

```
def forward(im, t):
  # ===== your code here! =====

  # TODO:
  # implement the forward process for timestep t
  alpha=alphas_cumprod[t]
  epsilon=torch.randn_like(im)
  noisy_im=torch.sqrt(alpha)*im+torch.sqrt(1-alpha)*epsilon

  # ==== end of code ====
  return noisy_im

for t in [250, 500, 750]:
  im_noisy = forward(test_im, t)
  media.show_image(im_noisy[0].permute(1,2,0) / 2. + 0.5, title=f"noisy t={t}")
```

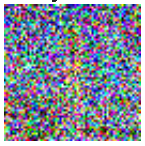
noisy t=250



noisy t=500



noisy t=750



1.2 Classical Denoising [TODO 2]

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

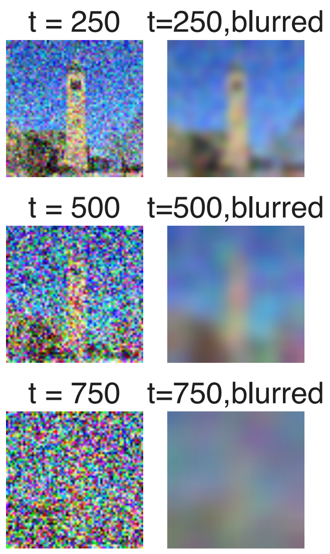
- Show the test image at noise level [250, 500, 750] from the previous part, for comparison
- Show the 3 noisy images filtered with Gaussian blur filtering

Hint

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

```
import torch.nn.functional as F
import torchvision.transforms.functional as TF

for t in [250, 500, 750]:
  # Dict of images to display
  im_dict = {}

  # Run forward process to get noisy image, and display
  im_noisy = forward(test_im, t)
  im_dict[f't = {t}'] = im_noisy[0].permute(1,2,0) / 2. + 0.5

  # Gaussian blur image
  # ===== your code here! =====
  if t==250:
      ksize=9
      sigma=2.0
  elif t==500:
      ksize=15
      sigma=4.0
  else:
      ksize=21
      sigma=6.0
  blurred_im=TF.gaussian_blur(im_noisy[0],kernel_size=ksize,sigma=sigma)
  im_dict[f't={t},blurred']=blurred_im.permute(1,2,0)/2.+0.5
```

```
# TODO:
# gaussian blur the image using TF.gaussian_blur
# add the result to im_dict to display

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

media.show_images(im_dict)
```



1.3 Implementing One Step Denoising [TODO 3]

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]):

- Using the UNet, denoise the image by estimating the noise.

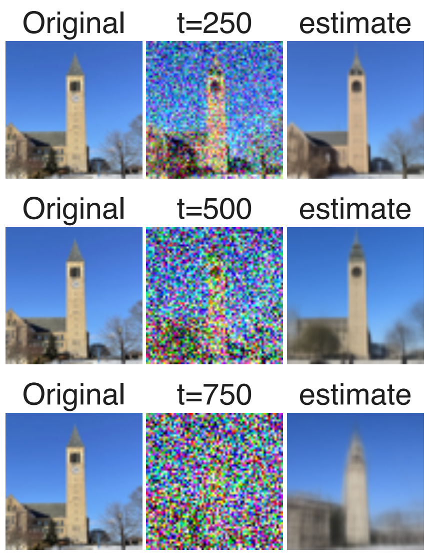
- Estimate the noise in the new noisy image, by passing it through `stage_1.unet`
- Remove the noise from the noisy image to obtain an estimate of the original image.
- Visualize the original image, the noisy image, and the estimate of the original image

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 as `half` precision (to save memory), so inputs to the denoiser will also need to be `half` precision.
- The signature for the unet is `stage_1.unet(image, 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 this prompt embedding
prompt_embeds = prompt_embeds_dict["a high quality photo"]

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

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

    # TODO:
```

```
    # create `im_noisy`, which is `test_im` passed through the forward process
    im_noisy=forward(test_im,t)

    # ==== 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
    # ===== your code here! =====

    # TODO:
    # create `clean_est`, the estimated clean image
    # also run `.detach().numpy()` on the image so we can display it
    clean_est=(im_noisy-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_c
    clean_est=clean_est.detach().cpu().numpy()
    # ==== end of code ====

    # Show the images
    media.show_images(
        {
            'Original': test_im[0].permute(1,2,0) / 2. + 0.5,
            f't={t}': im_noisy[0].permute(1,2,0) / 2. + 0.5,
            'estimate': clean_est[0].transpose(1,2,0) / 2. + 0.5,
        }
    )
```



1.4 Implementing Iterative Denoising [TODO 4]

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 noisiest image (and thus the largest t) and `strided_timesteps[-1]` will correspond to a clean image (and thus $t = 0$). 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:

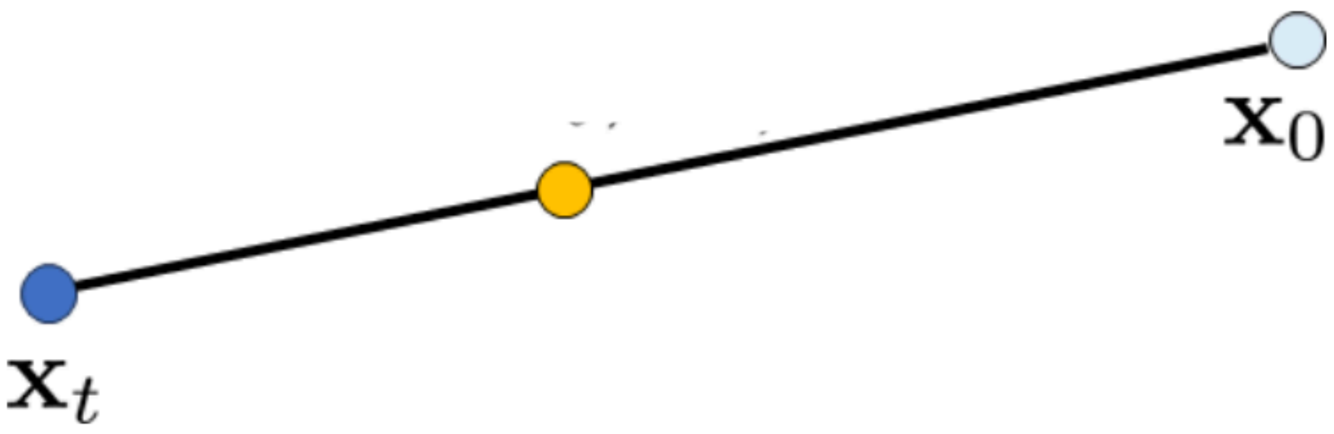
$$x_{t'} = \frac{\sqrt{\bar\alpha_{t'}}\beta_t}{1-\bar\alpha_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar\alpha_{t'})}{1-\bar\alpha_t}x_t + v_\sigma \tag{3}$$

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 using equation 2 just like in section 1.3

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. 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(image, i_start)`, which takes a noisy image `image`, 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 test image `im` 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 image 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.

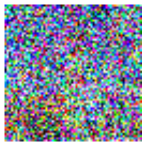
```
# Make timesteps. Must be list of ints satisfying:
# - monotonically decreasing
# - ends at 0
# - begins close to or at 999

# Make strided_timesteps
# ===== your code here! =====

# TODO:
# create `strided_timesteps`, a list of timesteps, from 990 to 0 in steps of 30
strided_timesteps=list(range(990,-1,-30))

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

stage_1.scheduler.set_timesteps(timesteps=strided_timesteps)    # Need this b/c v


def add_variance(predicted_variance, t, image):
  '''
  Args:
    predicted_variance : (1, 3, 64, 64) tensor, last three channels of the UNet o
    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_vari
  variance_noise = torch.randn_like(image)
  variance = torch.exp(0.5 * variance) * variance_noise
  return image + variance


def iterative_denoise(image, i_start, prompt_embeds, timesteps, display=True):
  with torch.no_grad():
    for i in range(i_start, len(timesteps) - 1):
      # Get timesteps
      t = timesteps[i]
      prev_t = timesteps[i+1]

      # Get alphas, betas
      # ===== your code here! =====

      # TODO:
```

```
      # get `alpha_cumprod` and `alpha_cumprod_prev` for timestep t from `alphas_
      alpha_cumprod=alphas_cumprod[t]
      alpha_cumprod_prev=alphas_cumprod[prev_t]
      alpha=alpha_cumprod/alpha_cumprod_prev
      beta=1-alpha
      # ==== end of code ====

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

      # Split estimate into noise and variance estimate
      noise_est, predicted_variance = torch.split(model_output, image.shape[1], d

      # Eq (6) and (7) of DDPM
      # ===== your code here! =====

      # TODO:
      # compute `pred_prev_image`, the DDPM estimate for the image at the
      # next timestep, which is slightly less noisy. Use the equation for
      # x_{t'} in the notes above.
      pred_x_0=(image-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_cum
      pred_prev_image = ((torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cum
                        ((torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - al
      pred_prev_image=add_variance(predicted_variance,prev_t,pred_prev_image)
      # ==== end of code ====

      # Show denoised image
      if i % 5 == 0 and display:
        media.show_images(
            {
                f'x_{t}': image.cpu()[0].permute(1,2,0) / 2. + 0.5,
            }
        )

      image = pred_prev_image

    clean = image.cpu().detach().numpy()

  return clean
```

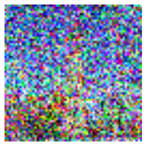
```
# Please use this prompt embedding
prompt_embeds = prompt_embeds_dict["a high quality photo"]

# Add noise
i_start = 10
t = strided_timesteps[i_start]
im_noisy = forward(test_im, t).half().to(device)

# Denoise
clean = iterative_denoise(im_noisy,
                          i_start=i_start,
                          prompt_embeds=prompt_embeds,
                          timesteps=strided_timesteps)

# One step denoise
# ===== your code here! =====
# TODO:
# Compute the one step estimate of the clean image. Feel free to copy and paste
# code from part 8.3. Store the image into `clean_one_step`.
with torch.no_grad():
    noise_est=stage_1.unet(
        im_noisy,
        t,
        encoder_hidden_states=prompt_embeds,
        return_dict=False
    )[0][:, :3]
    alpha_cumprod = stage_1.scheduler.alphas_cumprod[t]
    clean_one_step=(im_noisy-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(al
    clean_one_step=clean_one_step.cpu().detach().numpy()

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

# Gaussian blur denoise
# ===== your code here! =====

# TODO:
# Compute the gaussian blurred noisy image, using kernel_size=5 and sigma=2.
# Feel free to copy code from part 8.2. Store the image as `blur_filtered`
kernel_size = 5
sigma = 2
blur_filtered=TF.gaussian_blur(im_noisy[0],kernel_size=kernel_size,sigma=sigma)
blur_filtered=blur_filtered.unsqueeze(0)
# ==== end of code ====

# Show results
```

```
media.show_images(
    {
        'Original': test_im[0].cpu().permute(1,2,0) / 2. + 0.5,
        'clean': clean[0].transpose(1,2,0) / 2. + 0.5,  # clean is already a nump
        'clean 1 step': clean_one_step[0].transpose(1,2,0) / 2. + 0.5,  # already
        'gaussian': blur_filtered.cpu()[0].permute(1,2,0) / 2. + 0.5,
    }
)
```

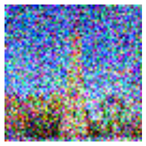
x_690



x_540



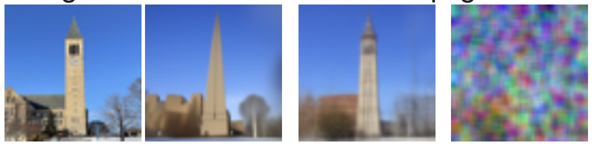
x_390



x_240

x_90

Original clean clean 1 step gaussian



1.5 Diffusion Model Sampling [TODO 5]

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 in random noise. This effectively denoises pure noise. Please do this, and show 5 results of `"a high quality photo"`.

Deliverables

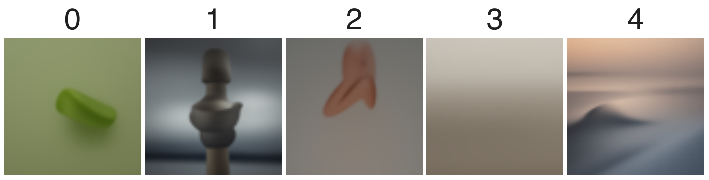
- Show 5 sampled images

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"]

generated = {}

for i in tqdm(range(5)):
  # Make random noise
  # ===== your code here! =====

  # TODO:
  # Make `noise`, random gaussian noise of shape (1, 3, 64, 64)
  noise=torch.randn(1,3,64,64).half().to(device)
  im_noisy=noise
  # ==== end of code ====

  # Denoise
  clean = iterative_denoise(im_noisy,
                            i_start=0,
                            prompt_embeds=prompt_embeds,
                            display=False,
                            timesteps=strided_timesteps)

  # Add to dict to display
  generated[i] = clean[0].transpose(1,2,0) / 2. + 0.5

# Show results
media.show_images(
  generated
)
```

```
100%|██████████| 5/5 [00:08<00:00,  1.73s/it]
```



1.6 Classifier Free Guidance [TODO 6]

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 and use `"a high quality photo"` for unconditional generation.

Deliverables

- Implement the `iterative_denoise_cfg` function
- Show 5 images of `"a high quality photo"` with a CFG scale of $\gamma = 7$

Hints

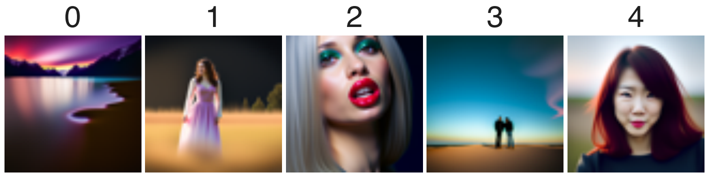
- You will need to run the UNet twice, once for the conditional prompt embedding, and once for the unconditional
- The UNet will predict both a conditional and an unconditional variance. Just use the conditional variance with the `add_variance` function.
- The resulting images should be much better than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

```
# The condition prompt embedding
prompt_embeds = prompt_embeds_dict['a high quality photo']

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict['']
```

```
def iterative_denoise_cfg(image, i_start, prompt_embeds, uncond_prompt_embeds, ti
  with torch.no_grad():
    for i in range(i_start, len(timesteps) - 1):
      # Get timesteps
      t = timesteps[i]
      prev_t = timesteps[i+1]

      # Get alphas, betas
      # ===== your code here! =====

      # TODO:
      # Get `alpha_cumprod`, `alpha_cumprod_prev`, `alpha`, `beta`
      # Feel free to copy code from part 8.4
      alpha_cumprod=alphas_cumprod[t]
      alpha_cumprod_prev=alphas_cumprod[prev_t]
      alpha =alpha_cumprod/alpha_cumprod_prev
      beta =1-alpha
      # ==== end of code ====

      # Get cond noise estimate
      model_output = stage_1.unet(
          image,
          t,
          encoder_hidden_states=prompt_embeds,
          return_dict=False
      )[0]

      # Get uncond noise estimate
      uncond_model_output = stage_1.unet(
          image,
          t,
          encoder_hidden_states=uncond_prompt_embeds,
          return_dict=False
      )[0]

      # Split estimate into noise and variance estimate
      noise_est, predicted_variance = torch.split(model_output, image.shape[1], d
      uncond_noise_est, _ = torch.split(uncond_model_output, image.shape[1], dim=

      # Do classifier free guidance
      # ===== your code here! =====

      # TODO:
```

```
      # Compute the CFG noise estimate and put it in `model_output`.
      # Hint: use `model_output` and `uncond_model_output`. Should only require
      # one line of code
      noise_est=uncond_noise_est+scale*(noise_est-uncond_noise_est)

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

      # Eq (6) and (7) of DDPM
      # ===== your code here! =====

      # TODO:
      # Get `pred_prev_image`, the next less noisy image.
      # Feel free to copy code from part 8.4
      pred_x_0=(image-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_cum
      pred_prev_image = ((torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cum
                        ((torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - al
      pred_prev_image=add_variance(predicted_variance,prev_t,pred_prev_image)
      # ==== end of code ====

      # Show denoised image
      if i % 5 == 0 and display:
        media.show_images(
            {
                f'x_{t}': image.cpu()[0].permute(1,2,0) / 2. + 0.5,
            }
        )

      image = pred_prev_image

    clean = image.cpu().detach().numpy()

  return clean

generated = {}
for i in tqdm(range(5)):
  # Make random noise
  im_noisy = torch.randn(1,3,64,64).half().to(device)

  # Denoise
  clean = iterative_denoise_cfg(im_noisy,
                                i_start=0,
                                prompt_embeds=prompt_embeds,
                                uncond_prompt_embeds=uncond_prompt_embeds,
                                display=False,
                                timesteps=strided_timesteps)
  # Add to dict to display
  generated[i] =clean[0].transpose(1,2,0)/2.+ 0.5

# Show results
media.show_images(
  generated
)
```

```
100%|██████████| 5/5 [00:16<00:00,  3.33s/it]
```



1.7 Image-to-image Translation [TODO 7]

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 test 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 test image (with a low-enough noise level). This follows the SDEdit algorithm.

To start, please run the forward process to get a noisy test image, 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

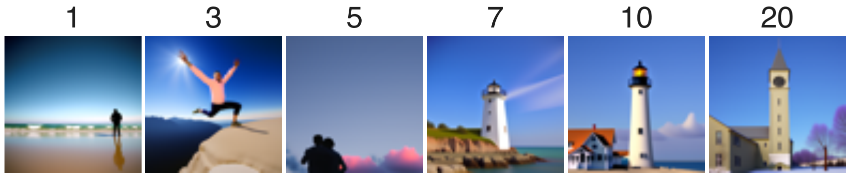
- Edits of the test image, using the given prompt at noise levels [1, 3, 5, 7, 10, 20] with text prompt `"a high quality photo"`
- Edits of 2 of your own test images, using the same procedure.

Hints

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

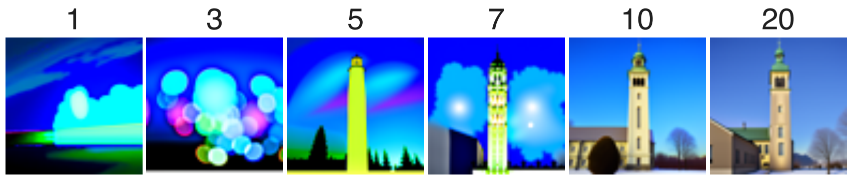
```
# Please use this prompt, as an "unconditional" text prompt
prompt_embeds = prompt_embeds_dict["a high quality photo"]

imgs_dict = {}
for i_start in tqdm([1, 3, 5, 7, 10, 20]):
  # Add noise
  t = strided_timesteps[i_start]
  # ===== your code here! =====

  # TODO:
  # Run the forward process on `test_im` and write the result to
  # `im_noisy`.
  im_noisy=forward(test_im,t).half().to(device)

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

  # 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[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

media.show_images(imgs_dict)
```

```
100%|██████████| 6/6 [00:15<00:00,  2.62s/it]
```

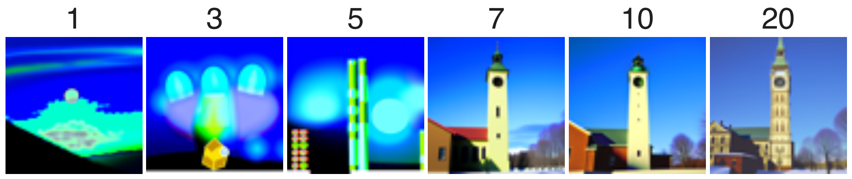


```
# Edits of 2 of your own test images, using the same procedure

your_images = ["img1.png", "img2.png"]  # Replace with your own image filenames
prompt_embeds = prompt_embeds_dict["a high quality photo"]

for image_path in your_images:
    print(f"Processing: {image_path}")
    # Load and preprocess image
    # (Convert to RGB, resize to 64x64, normalize to [-1, 1])
    img = Image.open(image_path).convert("RGB").resize((64, 64))
    img_tensor = TF.to_tensor(img)
    img_tensor = 2 * img_tensor - 1
    img_tensor = img_tensor[None]  # Add batch dimension

    imgs_dict = {}
    for i_start in [1, 3, 5, 7, 10, 20]:
        # ===== your code here! =====

        # TODO:
        # Run the forward process on your test image and write the result to
        # `im_noisy`.
        im_noisy=forward(test_im,t).half().to(device)

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

        clean = iterative_denoise_cfg(
            im_noisy,
            i_start=i_start,
            prompt_embeds=prompt_embeds,  # prompt = "a high quality photo"
            uncond_prompt_embeds=uncond_prompt_embeds,
            timesteps=strided_timesteps,
            display=False
        )
        imgs_dict[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

    # Display results for this image
    media.show_images(imgs_dict)
```

```
Processing: img1.png
```



```
Processing: img2.png
```



1.7.1 Editing Hand-Drawn and Web Images [TODO 7.1]

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
      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize to rang
  ])

  # 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
```

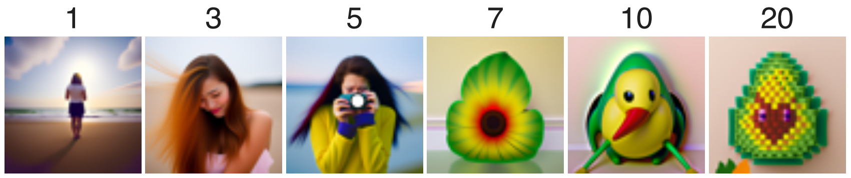
Download Images from Web

```
# @title Download Images from Web

################
## CHANGE URL ##
################
url = "https://i.pinimg.com/originals/76/e5/d5/76e5d55d0c8c6ec65135b42a2c5cbd98.j
################
################

# 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
```

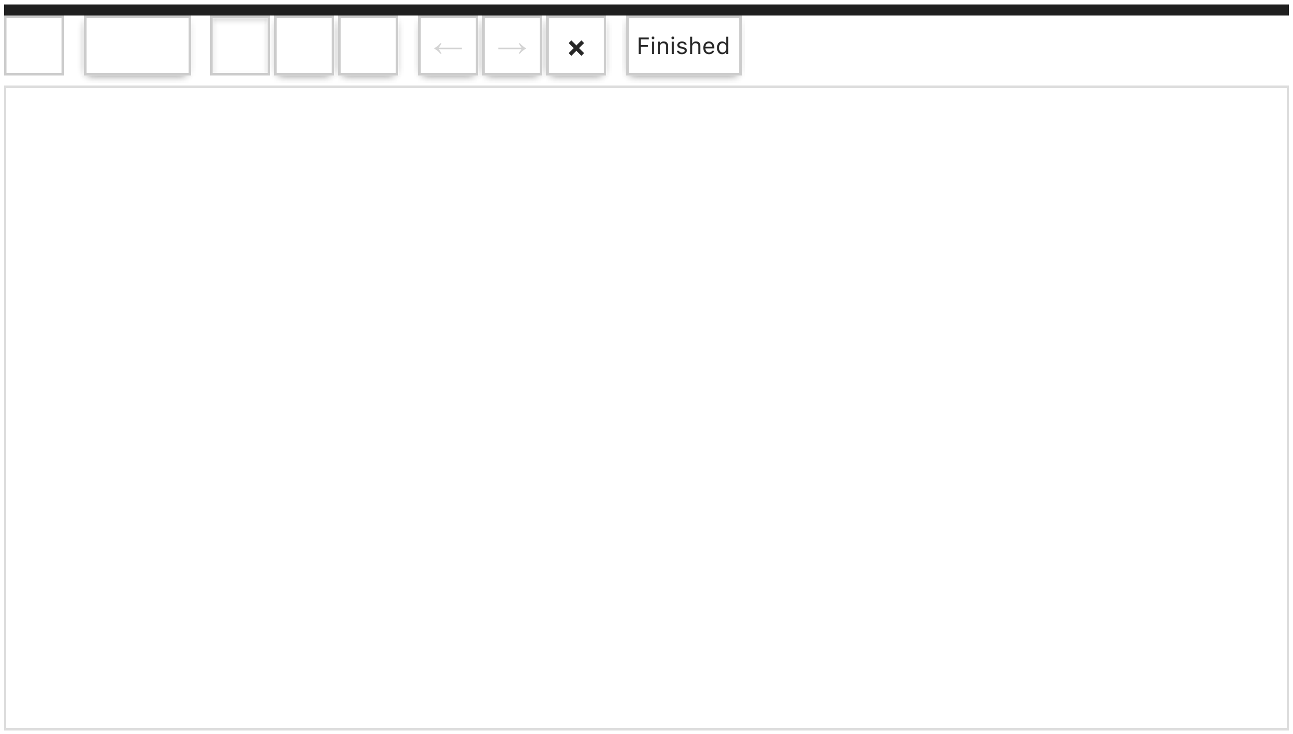
```
prompt_embeds = prompt_embeds_dict["a high quality photo"]

imgs_dict = {}
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_dict[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

media.show_images(imgs_dict)
```

```
100%|██████████| 6/6 [00:15<00:00,  2.58s/it]
```



Hand Drawn Images

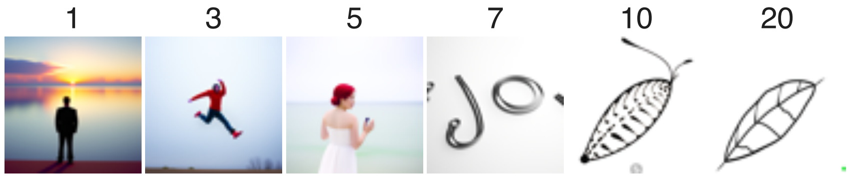
```
# @title Hand Drawn Images

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

# This code is from: https://gist.github.com/karim23657/5ad5e067c1684dbc76c93bd88
# 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/karim2365

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)
```



```
Processed image
```

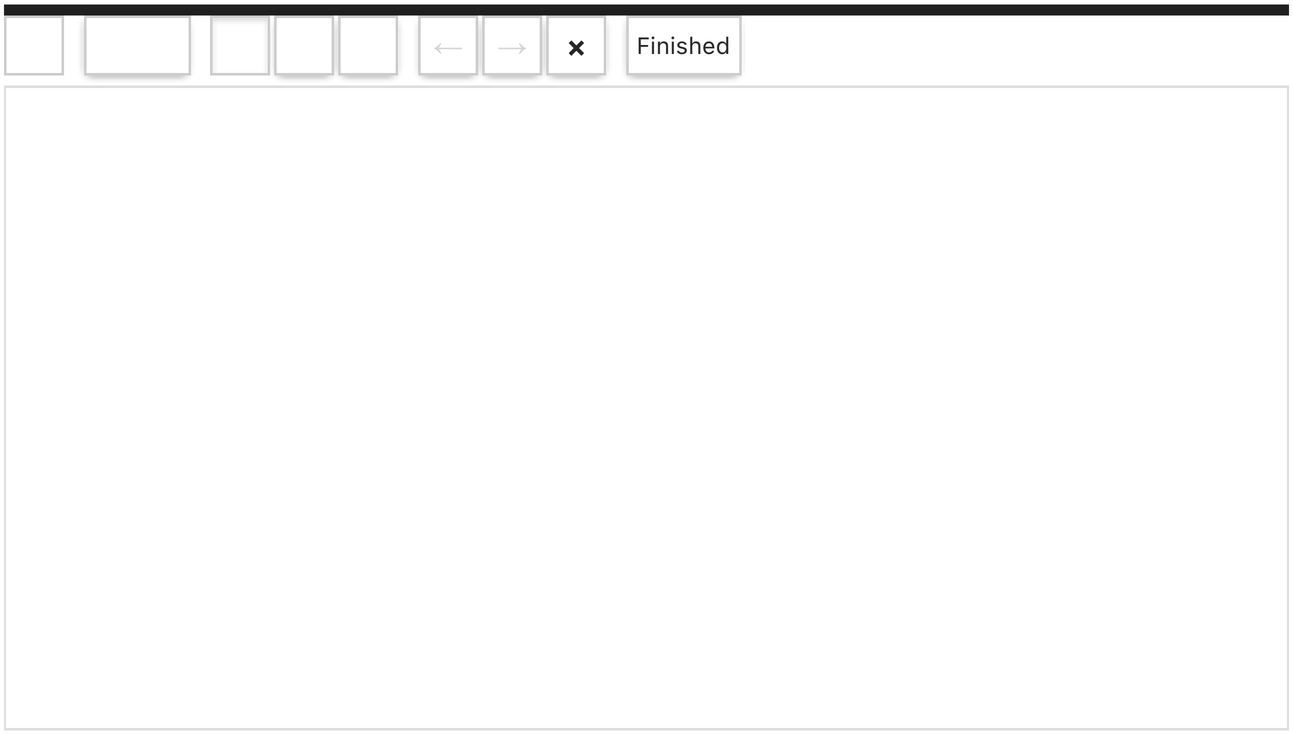
```
prompt_embeds = prompt_embeds_dict["a high quality photo"]

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[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

media.show_images(imgs_dict)
```

```
100%|██████████| 6/6 [00:15<00:00,  2.56s/it]
```



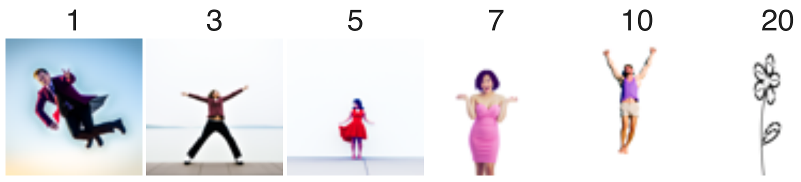
Second Hand Drawn Image

```
# @title Second Hand Drawn Image
drawn_im_2 = draw('myImage_2.png').resize((64,64))
drawn_im_2 = process_pil_im(drawn_im_2)

prompt_embeds = prompt_embeds_dict["a high quality photo"]

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_2, 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[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

media.show_images(imgs_dict)
```



```
Processed image
```

```
100%|██████████| 6/6 [00:15<00:00,  2.55s/it]
```



1.7.2 Inpainting [TODO 7.2]

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

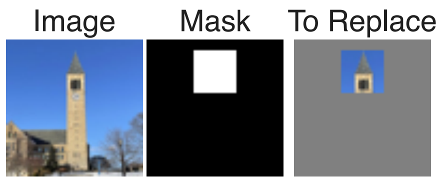
- A properly implemented `inpaint` function
- The test image inpainted (feel free to use your own mask)
- 2 of your own images edited (come up with your own mask)
  - look at the results from this paper for inspiration

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[:, :, 5:25, 22:42] = 1.0
mask = mask.to(device)
mask = mask.to(device).half()

# 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,
})
```

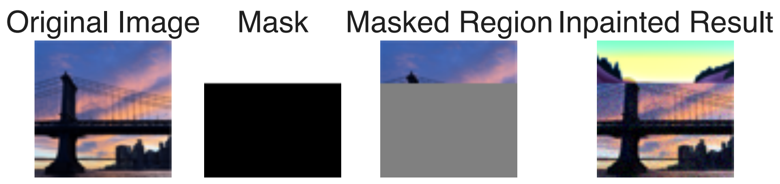


```
def inpaint(original_image, mask, prompt_embeds, uncond_prompt_embeds, timesteps,
  image = torch.randn_like(original_image).to(device).half()

  with torch.no_grad():
    for i in range(len(timesteps) - 1):
      # Get timesteps
      t = timesteps[i]
      prev_t = timesteps[i+1]

      # Get alphas, betas
      # ===== your code here! =====


      # TODO:
      # Get `alpha_cumprod`, `alpha_cumprod_prev`, `alpha`, `beta`
      # Feel free to copy code from part 8.4
      alpha_cumprod=alphas_cumprod[t]
      alpha_cumprod_prev=alphas_cumprod[prev_t]
      alpha=alpha_cumprod/alpha_cumprod_prev
      beta=1-alpha
      # ==== end of code ====

      # Get cond noise estimate
      model_output = stage_1.unet(
          image,
          t,
          encoder_hidden_states=prompt_embeds,
          return_dict=False
      )[0]

      # Get uncond noise estimate
      uncond_model_output = stage_1.unet(
          image,
          t,
          encoder_hidden_states=uncond_prompt_embeds,
          return_dict=False
      )[0]

      # Split estimate into noise and variance estimate
      noise_est, predicted_variance = torch.split(model_output, image.shape[1], d
      uncond_noise_est, _ = torch.split(uncond_model_output, image.shape[1], dim=

      # Do classifier free guidance
      # ===== your code here! =====

      # TODO:
      # Compute the CFG noise estimate and put it in `model_output`.
      # Feel free to copy from part 8.6
      noise_est=uncond_noise_est+scale*(noise_est-uncond_noise_est)

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

      # Eq (6) and (7) of DDPM
      # ===== your code here! =====

      # TODO:
      # Get `pred_prev_image`, the next less noisy image.
      # Feel free to copy code from part 8.4
      pred_x_0=(image-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_cum
      pred_prev_image = ((torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cum
                        ((torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - al
      pred_prev_image=add_variance(predicted_variance,prev_t,pred_prev_image)
      # ==== end of code ====

      # Show denoised image
      if i % 5 == 0 and display:
        media.show_images(
            {
                f'x_{t}': image.cpu()[0].permute(1,2,0) / 2. + 0.5,
            }
        )

      # Repaint - replace unmasked pixels with known pixels
      # ===== your code here! =====

      # TODO:
      # Update `pred_prev_image` as explained in the instructions.
      # You can reuse the `forward` function defined in part 8.1
      noisy_original=forward(original_image.half().to(device),t).half().to(device


      pred_prev_image=mask*pred_prev_image+(1-mask)*noisy_original
      # ==== end of code ====

      image = pred_prev_image

    clean = image.cpu().detach().numpy()

  return clean


# The condition prompt embedding
prompt_embeds = prompt_embeds_dict['a pencil']

# The unconditional prompt embedding
uncond_prompt_embeds = prompt_embeds_dict['']

# Denoise
inpainted = inpaint(original_image=test_im,
                    mask=mask,
                    prompt_embeds=prompt_embeds,
                    uncond_prompt_embeds=uncond_prompt_embeds,
                    timesteps=strided_timesteps,
                    display=False)

# Show results
media.show_images(
    {
        'inpainted': inpainted[0].transpose(1,2,0) / 2. + 0.5,
    }
)
```

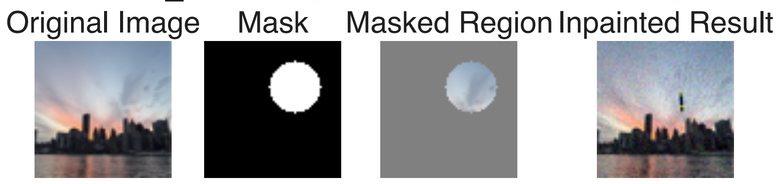
inpainted

```
# [TODO] Inpaint Two of Your Own Images with Custom Masks and Prompts

your_images = ["img1.png", "img2.png"]

for image_path in your_images:
    print(f"Processing: {image_path}")

    # === Load and preprocess image ===
    img = Image.open(image_path).convert("RGB").resize((64, 64))
    img_tensor = TF.to_tensor(img)
    img_tensor = 2 * img_tensor - 1
    img_tensor = img_tensor[None].to(device)

    # === [TODO] Define a custom mask and prompt for this image ===
    mask=torch.zeros_like(img_tensor)

    if image_path == your_images[0]:
        mask[:,:,0:20,:]=1.0
        mask=mask.to(device).half()
        prompt_embeds=prompt_embeds_dict["a lithograph of waterfalls"]
        uncond_prompt_embeds=prompt_embeds_dict[""]
    elif image_path == your_images[1]:
        h,w=img_tensor.shape[2],img_tensor.shape[3]
        center_y,center_x=h//3,2*w//3
        radius=min(h, w)//5
        y_grid,x_grid=torch.meshgrid(torch.arange(h),torch.arange(w))
        y_grid,x_grid=y_grid.to(device),x_grid.to(device)
        distance_from_center=torch.sqrt((y_grid-center_y)**2+(x_grid-center_x)**2
        circle_mask=distance_from_center<=radius
        for c in range(mask.shape[1]):
            mask[0,c]=circle_mask.float()
        mask=mask.to(device).half()



        prompt_embeds=prompt_embeds_dict["a pencil"]
        uncond_prompt_embeds=prompt_embeds_dict[""]

    # === Inpaint ===
    inpainted = inpaint(
        original_image=img_tensor,
        mask=mask,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        display=False
    )

    # === Visualize ===
    media.show_images({
        "Original Image": img_tensor[0].cpu().permute(1, 2, 0) / 2. + 0.5,
        "Mask": mask[0].cpu().permute(1, 2, 0),
        "Masked Region": (img_tensor * mask)[0].cpu().permute(1, 2, 0) / 2. + 0.5
        "Inpainted Result": inpainted[0].transpose(1, 2, 0) / 2. + 0.5,
    })
```

```
Processing: img1.png
```



```
Processing: img2.png
/usr/local/lib/python3.11/dist-packages/torch/functional.py:539: UserWarning:
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



1.7.3 Text-Conditioned Image-to-image Translation [TODO 7.3]

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 the precomputed prompts we provide you (if you want to use your own prompts, see appendix).

Deliverables

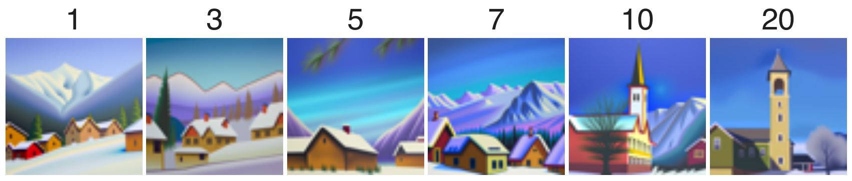
- Edits of the test image, using the given prompt at noise levels [1, 3, 5, 7, 10, 20]
- Edits of 2 of your own test images, using the same procedure.

Hints

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

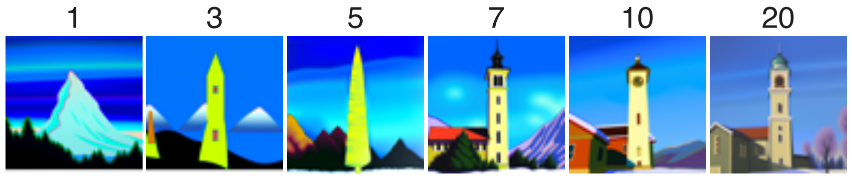
```
# Please use this prompt
prompt_embeds = prompt_embeds_dict['an oil painting of a snowy mountain village']

imgs_dict = {}
for i_start in tqdm([1, 3, 5, 7, 10, 20]):
  # Add noise
  t = strided_timesteps[i_start]
  # ===== your code here! =====

  # TODO:
  # Run the forward process in `test_im` and save the result to
  # `im_noisy`. This should be identical to the above
  im_noisy=forward(test_im,t).half().to(device)

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

  # 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[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

media.show_images(imgs_dict)
```

```
100%|██████████| 6/6 [00:15<00:00,  2.52s/it]
```

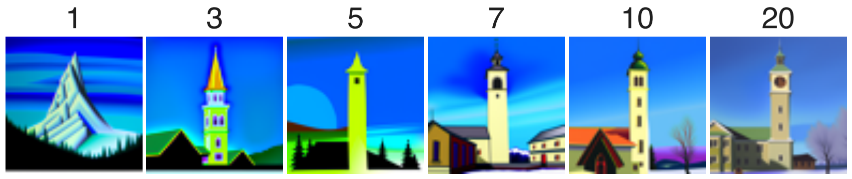


```
# [TODO] Edits of 2 of your own test images, using the same procedure

your_images = ["img1.png", "img2.png"]  # Replace with your own image filenames
prompt_embeds = prompt_embeds_dict['an oil painting of a snowy mountain village']

for image_path in your_images:
    print(f"Processing: {image_path}")
    # Load and preprocess image
    # (Convert to RGB, resize to 64x64, normalize to [-1, 1])
    img = Image.open(image_path).convert("RGB").resize((64, 64))
    img_tensor = TF.to_tensor(img)
    img_tensor = 2 * img_tensor - 1
    img_tensor = img_tensor[None]  # Add batch dimension

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

    imgs_dict = {}
    for i_start in [1, 3, 5, 7, 10, 20]:
        # ===== your code here! =====

        # TODO:
        # Run the forward process on your test image and write the result to
        # `im_noisy`. This should be identical to the above
        im_noisy=forward(test_im,t).half().to(device)

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

        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
        )
        imgs_dict[i_start] = clean[0].transpose(1,2,0) / 2. + 0.5

    # Display results for this image
    media.show_images(imgs_dict)
```

```
Processing: img1.png
```



```
Processing: img2.png
```



1.8 Visual Anagrams [TODO 8]

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 `"an oil painting of an old man"`, to obtain noise estimate ϵ_1. But at the same time, we will flip x_t upside down, and denoise with the prompt `"an oil painting of people around a campfire"`, to get noise estimate ϵ_2. We can flip ϵ_2 back, to make it right-side up, and average the two noise estimates. We can then perform a reverse diffusion step with the averaged noise estimate.

The full algorithm is:

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

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

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

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
- A visual anagram where on one orientation `"an oil painting of people around a campfire"` is displayed and, when flipped, `"an oil painting of an old man"` is displayed.
- 2 more illusions of your choice that change appearance when you flip it upside down (see the appendix if you want to generate your own prompts)

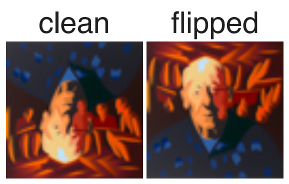
Hints

- 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, prompt_embeds, uncond_prompt_embeds, times
  with torch.no_grad():
    for i in range(i_start, len(timesteps) - 1):
      # Get timesteps
      t = timesteps[i]
      prev_t = timesteps[i+1]

      # Get alphas, betas
      # ===== your code here! =====

      # TODO:
      # Get `alpha_cumprod`, `alpha_cumprod_prev`, `alpha`, `beta`
      # Feel free to copy code from part 8.4
      alpha_cumprod=alphas_cumprod[t]
      alpha_cumprod_prev=alphas_cumprod[prev_t]
      alpha=alpha_cumprod/alpha_cumprod_prev
      beta=1-alpha
      # ==== end of code ====

      # Pair image with flipped image
      # ===== your code here! =====

      # TODO:
      # Create `input_images`, a (2, 3, 64, 64) tensor of two images to feed
      # into the diffusion model. One being `image` the noisy image and the
      # other being `image` but flipped upside down. `torch.flip` and
      # `torch.cat` will be useful here.
      flipped_image=torch.flip(image,dims=[2])
      input_images=torch.cat([image,flipped_image],dim=0)

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

      # Get cond noise estimate
      model_output = stage_1.unet(
          input_images,
          t,
          encoder_hidden_states=prompt_embeds,
          return_dict=False
      )[0]

      # Get uncond noise estimate
      uncond_model_output = stage_1.unet(
          input_images,
          t,
          encoder_hidden_states=uncond_prompt_embeds,
          return_dict=False
      )[0]

      # Flip output of denoiser
      # ===== your code here! =====

      # TODO:
      # Flip the outputs of the model right-side-up. You will need to change
      # both `model_output` and `uncond_model_output`. Again, `torch.flip`
      # will be useful, but pay special attention to dimensions.
      model_output = torch.cat([
          model_output[:1],
          torch.flip(model_output[1:2],dims=[2])
      ],dim=0)

      uncond_model_output = torch.cat([
          uncond_model_output[:1],
          torch.flip(uncond_model_output[1:2],dims=[2])
      ],dim=0)
      # ==== end of code ====

      # Split estimate into noise and variance estimate
      noise_est, predicted_variance = torch.split(model_output, image.shape[1], d
      uncond_noise_est, _ = torch.split(uncond_model_output, image.shape[1], dim=

      # Do classifier free guidance
      # ===== your code here! =====

      # TODO:
      # Compute the CFG noise estimate and put it in `noise_est`.
      # Feel free to copy from part 8.6
      noise_est=uncond_noise_est+scale*(noise_est-uncond_noise_est)

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

      # Take average of CFG'd noise estimates
      # ===== your code here! =====

      # TODO:
      # Update `model_output`, which should be of shape (2, 3, 64, 64)
      # representing 2 noise estimates, by averaging the two noise estaimtes,
      # resulting in a (1, 3, 64, 64) noise estiamte
      noise_est=noise_est.mean(dim=0,keepdim=True)
```

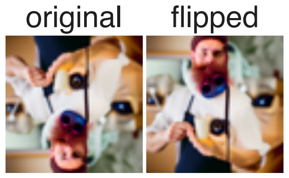
```
      # ==== end of code ====

      # Get variance (just use the variance of the first estimate)
      predicted_variance = predicted_variance[:1]

      # Eq (6) and (7) of DDPM
      # ===== your code here! =====

      # TODO:
      # Get `pred_prev_image`, the next less noisy image.
      # Feel free to copy code from part 8.4
      pred_x_0=(image-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_cum
      pred_prev_image = ((torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cum
                        ((torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - al
      pred_prev_image=add_variance(predicted_variance,prev_t,pred_prev_image)

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

      # Show denoised image
      if i % 5 == 0 and display:
        media.show_images(
            {
                f'x_{t}': image.cpu()[0].permute(1,2,0) / 2. + 0.5,
            }
        )

      image = pred_prev_image

    clean = image.cpu().detach().numpy()

  return clean

# We will use these prompt embeddings
prompt_embeds = torch.cat([
    prompt_embeds_dict['an oil painting of people around a campfire'],
    prompt_embeds_dict['an oil painting of an old man'],
])

uncond_prompt_embeds = torch.cat([
    prompt_embeds_dict[''],
    prompt_embeds_dict[''],
])

# Make random noise
im_noisy = torch.randn(1,3,64,64).half().to(device)
```

```
# Denoise
clean = make_flip_illusion(im_noisy,
                           i_start=0,
                           prompt_embeds=prompt_embeds,
                           uncond_prompt_embeds=uncond_prompt_embeds,
                           timesteps=strided_timesteps,
                           display=False)

# Show results
media.show_images(
    {
        'clean': clean[0].transpose(1,2,0) / 2. + 0.5,
        'flipped': clean[0].transpose(1,2,0)[::-1] / 2. + 0.5,
    }
)
```

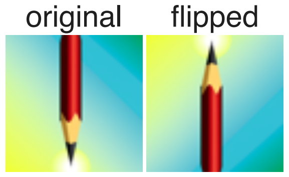


```
# [TODO] Create 2 more flip illusions of your choice
# Replace the prompts below with your own creative pairs!

your_prompt_pairs = [
    ("a photo of a dog", "a photo of a hipster barista"),
    ("a rocket ship", "a pencil"),
]

for idx, (prompt_top, prompt_bottom) in enumerate(your_prompt_pairs):
    print(f"\nGenerating Flip Illusion {idx+1}: '{prompt_top}' + '{prompt_bottom}

    prompt_embeds = torch.cat([
        prompt_embeds_dict[prompt_top],
        prompt_embeds_dict[prompt_bottom],
    ])

    uncond_prompt_embeds = torch.cat([
        prompt_embeds_dict[''],
        prompt_embeds_dict[''],
    ])
    # Generate noise
    im_noisy = torch.randn(1, 3, 64, 64).half().to(device)

    # Generate flipped illusion
    clean = make_flip_illusion(
        im_noisy,
        i_start=0,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        display=False
    )

    # Display original and flipped
    media.show_images({
        'original': clean[0].transpose(1, 2, 0) / 2. + 0.5,
        'flipped': clean[0].transpose(1, 2, 0)[::-1] / 2. + 0.5,
    })
```

```
Generating Flip Illusion 1: 'a photo of a dog' + 'a photo of a hipster barista
```



```
Generating Flip Illusion 2: 'a rocket ship' + 'a pencil'
```



1.9 Hybrid Images [TODO 9]

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:

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

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

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

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

- Correctly implemented `make_hybrids` function
- An image that looks like a `skull` from far away but a `waterfall` from close up
- 2 more hybrid images of your choosing.

Hints

- Please use `torchvision.transforms.functional.gaussian_blur`. Here is the documentation.
- You may have to run multiple times to get a really good result for the same reasons as above

Run the following cell to get the correct prompt embeddings:

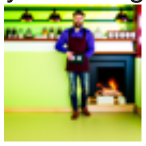
```
def make_hybrids(image, i_start, prompt_embeds, uncond_prompt_embeds, timesteps,
  with torch.no_grad():
    for i in range(i_start, len(timesteps) - 1):
      # Get timesteps
      t = timesteps[i]
      prev_t = timesteps[i+1]

      # Get alphas, betas
      # ===== your code here! =====

      # TODO:
      # Get `alpha_cumprod`, `alpha_cumprod_prev`, `alpha`, `beta`
      # Feel free to copy code from part 8.4
      alpha_cumprod=alphas_cumprod[t]
      alpha_cumprod_prev=alphas_cumprod[prev_t]
      alpha=alpha_cumprod/alpha_cumprod_prev
      beta=1-alpha
      # ==== end of code ====

      # Duplicate image, to get two noise estimates
      # ===== your code here! =====

      # TODO:
      # Create `input_images`, a (2, 3, 64, 64) tensor of two images to feed
      # into the diffusion model. This should be two copies of the same
      # `image` tensor, resulting in estimating the noise on the same noisy
      # image, but with two different prompts. `torch.cat` will be useful here.

      input_images=torch.cat([image, image],dim=0)

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

      # Get cond noise estimate
      model_output = stage_1.unet(
          input_images,
          t,
          encoder_hidden_states=prompt_embeds,
          return_dict=False
      )[0]

      # Get uncond noise estimate
      uncond_model_output = stage_1.unet(
          input_images,
          t,
          encoder_hidden_states=uncond_prompt_embeds,
          return_dict=False
      )[0]
      # Split estimate into noise and variance estimate
      noise_est, predicted_variance = torch.split(model_output, image.shape[1], d
      uncond_noise_est, _ = torch.split(uncond_model_output, image.shape[1], dim=

      # Do classifier free guidance
      # ===== your code here! =====

      # TODO:
      # Compute the CFG noise estimate and put it in `noise_est`.
      # Feel free to copy from part 8.6
      noise_est=uncond_noise_est+scale*(noise_est-uncond_noise_est)

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

      # Construct noise estimate from CFG'd noise estimates
      # ===== your code here! =====

      # TODO:
      # Construct the "composite noise estimate" and overwrite `noise_est`
      # This new noise estimate should be the sum of a low pass of one
      # noise estimate and the high pass of the other noise estimate.
      # `TF.gaussian_blur` will be useful here.
      # --> Please low pass the first noise estimate and high pass
      # --> the second noise estimate!

      kernel_size=33
      sigma=2

      low_pass=TF.gaussian_blur(noise_est[0:1],kernel_size=kernel_size,sigma=sigm

      high_pass=noise_est[1:2]-TF.gaussian_blur(noise_est[1:2],kernel_size=kernel

      noise_est=low_pass+high_pass


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

      # Get variance
      predicted_variance = predicted_variance[:1]

      # Eq (6) and (7) of DDPM
      # ===== your code here! =====

      # TODO:
      # Get `pred_prev_image`, the next less noisy image.
      # Feel free to copy code from part 8.4
      pred_x_0=(image-torch.sqrt(1-alpha_cumprod)*noise_est)/torch.sqrt(alpha_cum
      pred_prev_image = ((torch.sqrt(alpha_cumprod_prev) * beta) / (1 - alpha_cum
                        ((torch.sqrt(alpha) * (1 - alpha_cumprod_prev)) / (1 - al
      pred_prev_image=add_variance(predicted_variance,prev_t,pred_prev_image)
      # ==== end of code ====

      # Show denoised image
      if i % 5 == 0 and display:
        media.show_images(
            {
                f'x_{t}': image.cpu()[0].permute(1,2,0) / 2. + 0.5,
            }
        )

      image = pred_prev_image

    clean = image.cpu().detach().numpy()

  return clean

# We will use these text prompts
prompt_embeds = torch.cat([
    prompt_embeds_dict['a lithograph of a skull'],
    prompt_embeds_dict['a lithograph of waterfalls'],
])

uncond_prompt_embeds = torch.cat([
    prompt_embeds_dict[''],
    prompt_embeds_dict[''],
])

# Make random noise
im_noisy = torch.randn(1,3,64,64).half().to(device)

# Denoise
clean = make_hybrids(im_noisy,
                     i_start=0,
                     prompt_embeds=prompt_embeds,
                     uncond_prompt_embeds=uncond_prompt_embeds,
                     timesteps=strided_timesteps,
                     display=False)

# Show results
media.show_images(
    {
        'clean': clean[0].transpose(1,2,0) / 2. + 0.5,
    }
)
```

clean

```
# [TODO] Create 2 More Hybrid Images of Your Choosing with the make_hybrids funct

# Define your own prompt pairs
your_prompt_pairs = [
    ("a photo of a dog", "a photo of a hipster barista"),
    ("a rocket ship", "a pencil"),
]

for prompt1, prompt2 in your_prompt_pairs:
    print(f"Generating hybrid: '{prompt1}' + '{prompt2}'")

    prompt_embeds = torch.cat([
        prompt_embeds_dict[prompt1],
        prompt_embeds_dict[prompt2],
    ])

    uncond_prompt_embeds = torch.cat([
        prompt_embeds_dict[''],
        prompt_embeds_dict[''],
    ])

    # Generate hybrid image from noise
    im_noisy = torch.randn(1, 3, 64, 64).half().to(device)
    clean = make_hybrids(
        im_noisy,
        i_start=0,
        prompt_embeds=prompt_embeds,
        uncond_prompt_embeds=uncond_prompt_embeds,
        timesteps=strided_timesteps,
        display=False
    )

    # Display result
    media.show_images({
        'Hybrid Image': clean[0].transpose(1, 2, 0) / 2. + 0.5,
    })
```

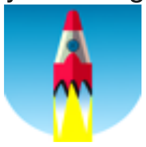
```
Generating hybrid: 'a photo of a dog' + 'a photo of a hipster barista'
```

Hybrid Image



```
Generating hybrid: 'a rocket ship' + 'a pencil'
```

Hybrid Image



Extra Credit: Create Something Cool!

Use what you've learned in this project to create something cool. **Please show your code and resulting image(s) in the cell below.**

Have fun!

```
# [Extra Credit] Show your code and results here!
```

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

[Optional] Using Your Own Text Prompts

Throughout this problem set, we have provided a dict of prompt embeddings so that you don't have to load the entire T5 text encoder on to the GPU. As part of your deliverable, you will need to come up with your own images for each part, and your own text prompts in the later parts as well.

To use your own text prompts, please use the following code to generate text embeddings. Be aware that downloading this model may take a few minutes, and be careful of GPU out-of-memory errors! If you'd like, you can pay for Colab Pro credits and use an A100 GPU which can load both the text model and diffusion model at once.

Loading T5 text encoder

```
# @title Loading T5 text encoder

# Load the T5 text encoder (this may take a while)
text_encoder = T5EncoderModel.from_pretrained(
    "DeepFloyd/IF-I-L-v1.0",
    subfolder="text_encoder",
    load_in_8bit=True,
    variant="8bit",
)
text_pipe = DiffusionPipeline.from_pretrained(
    "DeepFloyd/IF-I-L-v1.0",
    text_encoder=text_encoder,  # pass the previously instantiated text encoder
    unet=None
)
```

```
config.json: 100%                                    741/741 [00:00<00:00, 86.9kB/s]
The `load_in_4bit` and `load_in_8bit` arguments are deprecated and will be ren
model.8bit.safetensors:   1%                      115M/7.92G [00:04<05:17, 24.5MB/s]
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-86-10027b6cafd4> in <cell line: 0>()
      2 
      3 # Load the T5 text encoder (this may take a while)
----> 4 text_encoder = T5EncoderModel.from_pretrained(
      5     "DeepFloyd/IF-I-L-v1.0",
      6     subfolder="text_encoder",

                                   18 frames
/usr/lib/python3.11/ssl.py in read(self, len, buffer)
   1164         try:
   1165             if buffer is not None:
-> 1166                 return self._sslobj.read(len, buffer)
   1167             else:
   1168                 return self._sslobj.read(len)

KeyboardInterrupt: 
```

Making prompt embeds dict

```
# @title Making prompt embeds dict

# Prompts to put into dictionary
def get_prompt_embeds_dict():
  prompts = [
    'an oil painting of a snowy mountain village',
    'a photo of the amalfi cost',
    'a photo of a man',
    'a photo of a hipster barista',
    'a photo of a dog',
    'an oil painting of people around a campfire',
    'an oil painting of an old man',
    'a lithograph of waterfalls',
    'a lithograph of a skull',
    'a man wearing a hat',
    'a high quality photo',
    "a rocket ship",
    "a pencil"
    '',   # For CFG
  ]

  # Get prompt embeddings using the T5 model
  # each embedding is of shape [1, 77, 4096]
  # 77 comes from the max sequence length that deepfloyd will take
  # and 4096 comes from the embedding dimension of the text encoder
  prompt_embeds = [text_pipe.encode_prompt(prompt) for prompt in prompts]
  prompt_embeds, negative_prompt_embeds = zip(*prompt_embeds)
  prompt_embeds_dict = dict(zip(prompts, prompt_embeds))

  return prompt_embeds_dict
```

Exporting prompt embeds dict

```
prompt_embeds_dict = get_prompt_embeds_dict()

# @title Exporting prompt embeds dict

# Save prompt embeds to colab disk
torch.save(prompt_embeds_dict, 'prompt_embeds_dict.pth')

# Download prompt embeds
from google.colab import files
files.download('prompt_embeds_dict.pth')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-88-b9c30cdfb484> in <cell line: 0>()
----> 1 prompt_embeds_dict = get_prompt_embeds_dict()
      2 
      3 # @title Exporting prompt embeds dict
      4 
      5 # Save prompt embeds to colab disk

                          1 frames

<ipython-input-87-4c64804fa4ba> in <listcomp>(.0)
     24     # 77 comes from the max sequence length that deepfloyd will take
     25     # and 4096 comes from the embedding dimension of the text encoder
---> 26     prompt_embeds = [text_pipe.encode_prompt(prompt) for prompt in
prompts]
     27     prompt_embeds, negative_prompt_embeds = zip(*prompt_embeds)
     28     prompt_embeds_dict = dict(zip(prompts, prompt_embeds))

NameError: name 'text_pipe' is not defined
```

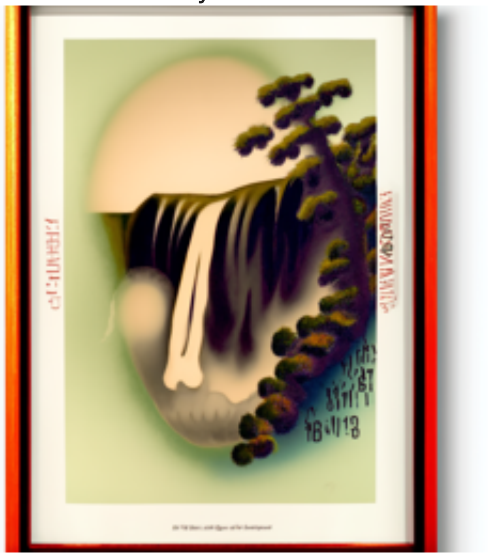
[Optional] Upsampling Images

If you want to upsample your 64×64 images to 256×256, you can use the following code:

```
def upsample(image64, prompt_embeds):
  """
  Args:
    image64 : torch tensor of size (1, 3, 64, 64) representing the image
    prompt_embeds : torch tensor of size (1, 77, 4096), prompt embedding
  Returns:
    image256 : torch tensor of size (1, 3, 256, 256), upsampled image
  """

  image256 = stage_2(
      image=image64,
      num_inference_steps=30,
      prompt_embeds=prompt_embeds,
      negative_prompt_embeds=prompt_embeds_dict[''],
      output_type="pt",
  ).images.cpu()

  return image256

hybrid_256 = upsample(torch.tensor(clean),
                      prompt_embeds[:1])
media.show_images(
    {
        'hybrid': hybrid_256[0].permute(1,2,0) / 2. + 0.5,
    }
)
```

```
100%                                          30/30 [00:18<00:00,  1.44it/s]
```

hybrid



Generating a PDF for CMSX

You can just use `File > Print` to get a pdf of this page. Please double check that no outputs are cutoff!