1. Verify the IF Deep Floyd model works [5 pts]

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. Experiment with different prompts and `num_inference_steps`. NOTE: if the upsampling in stage_2 is too slow, you can opt to only run and visualize stage_1.

Deliverables

- For the 3 text prompts in the provided code below, display the caption and the output of the model. Briefly comment on the quality of the outputs and their relationships to the text prompts.
- Choose one of the prompts and try varying the `num_inference_steps`, visualize the results for at least 2 other `num_inference_steps` values besides the deault.

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

# Retrieve embeddings, ensuring they exist in the dictionary
#prompt_embeds = torch.cat([prompt_embeds_dict.get(prompt, torch.zeros(1, 77, 4096)) for prompt in prompts], dim=0)
#negative_prompt_embeds = torch.cat([prompt_embeds_dict.get('', torch.zeros(1, 77, 4096))] * len(prompts))

print(prompt_embeds)

# 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
```

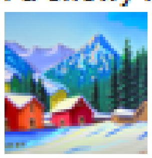
```python
# Sample from stage 2
# Outputs a [N, 3, 256, 256] torch tensor
# num_inference_steps is an integer between 1 and 1000, indicating how many
# denoising steps to take: lower is faster, at the cost of reduced quality
stage_2_output = stage_2(
    image=stage_1_output,
    num_inference_steps=20,
    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]
media.show_images(
    stage_1_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts)
media.show_images(
    stage_2_output.permute(0, 2, 3, 1).cpu() / 2. + 0.5,
    titles=prompts)
```

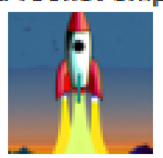
```
tensor([[[-0.1071, -0.0511, -0.0876,  ...,  0.0017, -0.0082,  0.0562],
         [-0.1246, -0.0109, -0.0011,  ..., -0.0404, -0.1946,  0.0525],
         [ 0.1464,  0.0292, -0.0322,  ..., -0.0244, -0.0499, -0.0247],
         ...,
         [ 0.0329,  0.1768, -0.2050,  ...,  0.1410, -0.0032, -0.0367],
         [ 0.0329,  0.1768, -0.2050,  ...,  0.1410, -0.0032, -0.0367],
         [ 0.0329,  0.1768, -0.2050,  ...,  0.1410, -0.0032, -0.0367]],

        [[-0.0577, -0.0786,  0.0555,  ...,  0.0726, -0.1164, -0.0948],
         [-0.2379, -0.0973,  0.0234,  ...,  0.1606, -0.0738, -0.1198],
         [-0.0742, -0.0978, -0.0535,  ..., -0.0347, -0.1416,  0.0022],
         ...,
         [ 0.0815,  0.1569, -0.0107,  ...,  0.2197, -0.1858,  0.0060],
         [ 0.0815,  0.1569, -0.0107,  ...,  0.2197, -0.1858,  0.0060],
         [ 0.0815,  0.1569, -0.0107,  ...,  0.2197, -0.1858,  0.0060]],

        [[-0.0434, -0.0486,  0.0099,  ..., -0.0546, -0.1963,  0.0199],
         [ 0.0209,  0.0127,  0.0989,  ...,  0.1277, -0.0831, -0.0284],
         [-0.0217,  0.0692,  0.0727,  ..., -0.0797,  0.1035, -0.0363],
         ...,
         [-0.0604,  0.2896, -0.0640,  ...,  0.1376, -0.0888,  0.0272],
         [-0.0604,  0.2896, -0.0640,  ...,  0.1376, -0.0888,  0.0272],
         [-0.0604,  0.2896, -0.0640,  ...,  0.1376, -0.0888,  0.0272]]],
       device='cuda:0', dtype=torch.float16)
```

100% 20/20 [00:07<00:00, 3.69it/s]

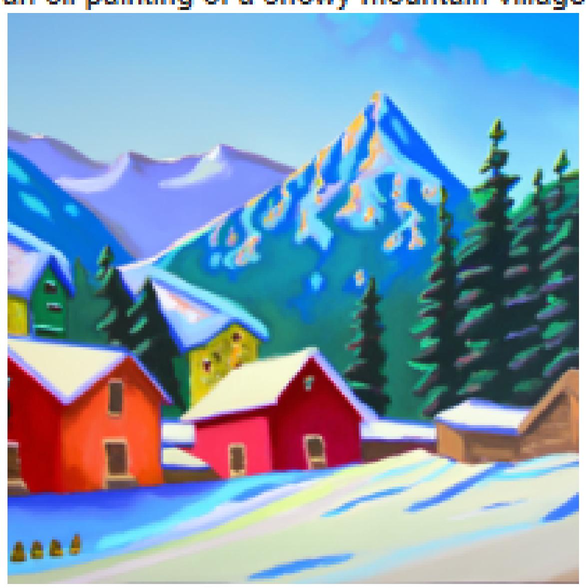
100% 20/20 [00:27<00:00, 1.39s/it]

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



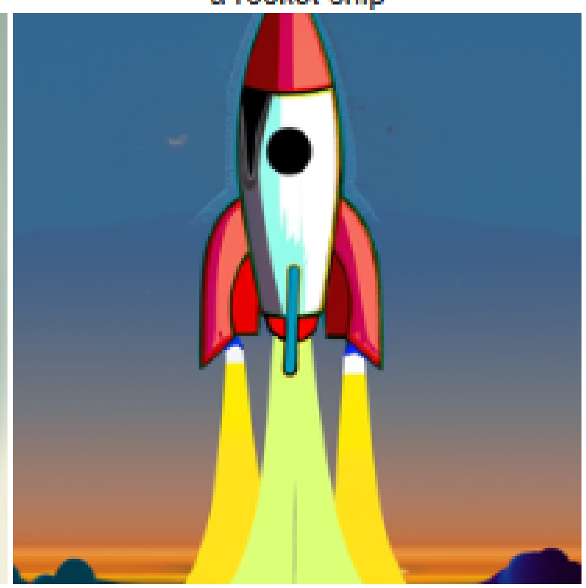


an oil painting of a snowy mountain village



a man wearing a hat

a rocket ship



2.1 Implementing the forward process [10pts]

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.

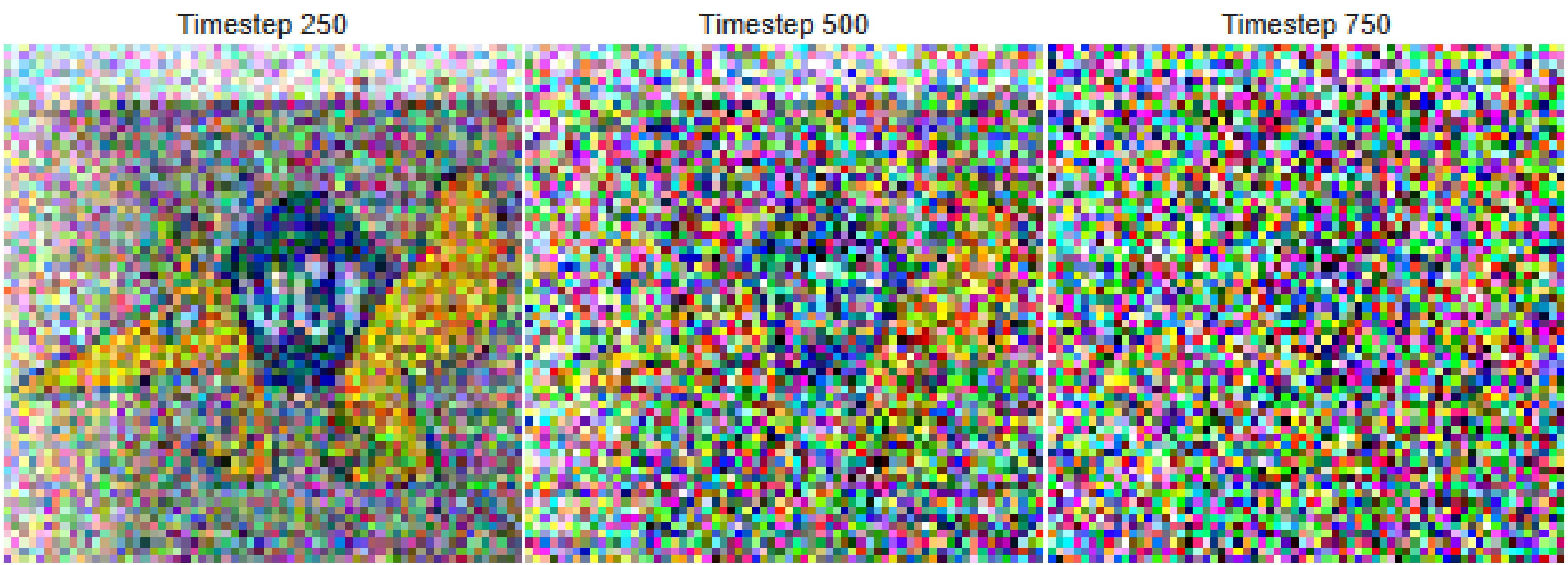
Delivarables

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

Hints

- The `torch.randn_like` function is helpful for creating gaussian noise ϵ the same size as a given tensor.
- Use the `alphas_cumprod` variable defined, which contains an array of the hyperparameters, with `alphas_cumprod[t]` corresponding to $\bar\alpha_t$.

```python
def forwardnoise(im, t):
  alphabar = alphas_cumprod[t]
  noise = torch.randn_like(im)
  im_noisy = torch.sqrt(alphabar) * im + torch.sqrt(1 - alphabar) * noise
  return im_noisy

#test code
todisplay = {}
for t in [250,500,750]:
  im_noisy = forwardnoise(test_im, t)
  todisplay[f"Timestep {t}"] = im_noisy[0].permute(1,2,0) / 2. + 0.5

#height,widht specify options to show_images to display the result at a larger dimension
# in the notebook even thought the image itself is 64x64
media.show_images(todisplay,width=256,height=256)
```



2.2 Classical Denoising by blurring [5pts]

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.

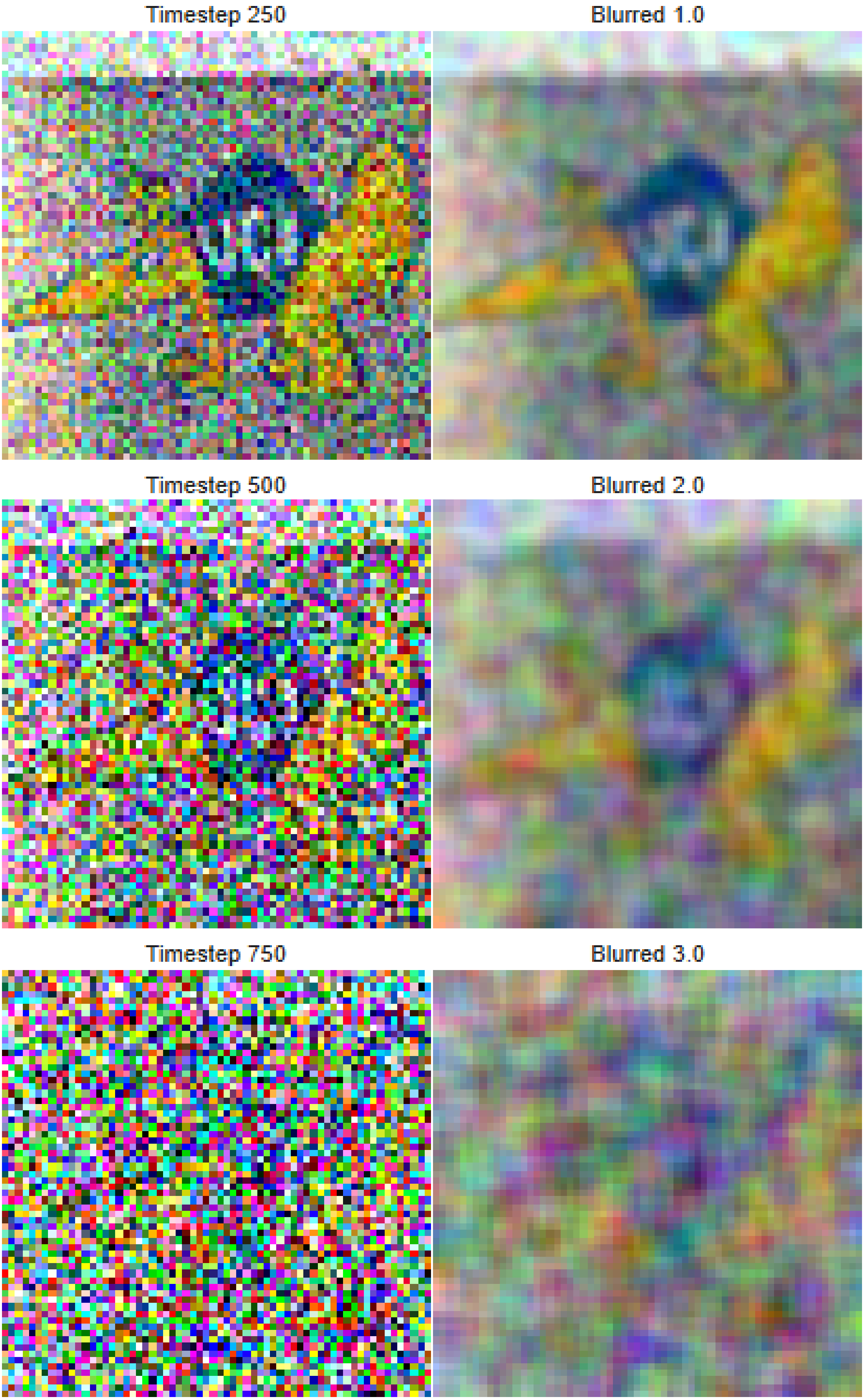
Deliverables

- For each of the 3 noisy test 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.
- media.show_images() will take a dict of images and display them in a row where the captions are the keys and the image data is the value (documentation)

[11]
```
tvals = [250,500,750]
svals = [1.0,2.0,3.0]
for i in range(3):
  t = tvals[i]
  s = svals[i]

  im_noisy = forwardnoise(test_im, t)
  blur = TF.gaussian_blur(im_noisy, kernel_size=[5, 5], sigma=[s, s])

  disp={}
  disp[f"Timestep {t}"] = im_noisy[0].permute(1, 2, 0) / 2. + 0.5  # Noisy image
  disp[f"Blurred {s}"] = blur[0].permute(1, 2, 0) / 2. + 0.5     # Blurred image

  media.show_images(disp,width=256,height=256)
```


Timestep 250
Blurred 1.0
Timestep 500
Blurred 2.0
Timestep 750
Blurred 3.0

2.3 Implementing One Step Denoising [10pts]

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 each of the 3 noisy images from 2.2 (wheree t = [250, 500, 750]) we will denoise the image by using the UNet to estimate 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 (above) the noise and x_0 are scaled. You will need to look at equation 2 to figure out how we can predict x_0 based on the noise estimate, 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 (since we aren't training the model), so this saves a considerable amount of memory.

[12]
```
# Please use this prompt embedding
prompt_embeds = prompt_embeds_dict["a high quality photo"]

with torch.no_grad():
  tvals = [250,500,750]
  for i in [0,1,2]:
    t = tvals[i]
    alphabar = alphas_cumprod[t]

    # Run forward process
    im_noisy = forwardnoise(test_im, t)

    # Estimate noise in noisy image
    noise_est = stage_1.unet(
        im_noisy.half().to(device),
        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
    clean_est = (im_noisy.cpu() - torch.sqrt(1 - alphabar) * noise_est) / torch.sqrt(alphabar)

    disp={}
    disp["Original"] = test_im[0].permute(1,2,0)/2.+0.5
    disp[f"Noisy {t}"] = im_noisy[0].permute(1, 2, 0) / 2. + 0.5  # Noisy image
    disp["Noise Estimate"] = noise_est[0].permute(1, 2, 0) / 2. + 0.5  # Noise estimate
    disp["DeNoised"] = clean_est[0].permute(1, 2, 0) / 2. + 0.5  # Denoised image
    media.show_images(disp,width=256,height=256)
```


Original
Noisy 250
Noise Estimate
DeNoised
Original
Noisy 500
Noise Estimate
DeNoised
Original
Noisy 750
Noise Estimate
DeNoised
UCI

2.1 Iterative Denoising Implementation [15pts]

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 go back try different "schedules" of timesteps.

Then 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 `t = timestep[i_start]`, applying the above formula to obtain an image at the earlier time `t' = timestep[i_start + 1]`, and repeat iteratively until we arrive at a clean image.

Add noise to the test image `im` to using your forward function with `t=timestep[10]` and display this noisy 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.

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 after the iterative denoising has completed
- Also show the predicted clean image using only a single large denoising step, as was done in the previous part. This should look much worse.

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 $\sigma(t)\epsilon$ 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.

[13]
```python
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
  '''
  # chekc for device
  t = t.to(stage_1.scheduler.timesteps.device)

  # 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

  # Clip the variance to prevent extreme values
  variance = torch.clamp(variance, -1.0, 1.0)

  return image + variance
```

[14]
```python
# 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
strided_timesteps = list(range(990, -1, -30))

# if necessary, force the last entry to be 0
if strided_timesteps[-1] != 0:
    strided_timesteps.append(0)

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

stage_1.scheduler.set_timesteps(timesteps=strided_timesteps)    # Need this for the variance computation to behave
print(strided_timesteps)
```

```
[990, 960, 930, 900, 870, 840, 810, 780, 750, 720, 690, 660, 630, 600, 570, 540, 510, 480, 450, 420, 390, 360, 330, 300, 270, 240, 210, 180, 150, 120, 90, 60, 30, 0]
```

Now to implement your iterative denoising process

```python
[65] def iterative_denoise(image, i_start, prompt_embeds, timesteps, display=True):
       with torch.no_grad():
         #tqdm will display a progress bar while running our for-loop
         for i in tqdm(range(i_start, len(timesteps) - 1),"iterative denoising"):

           image = image.half()
           # Get timesteps.
           # Since timesteps goes from T down to 0, prev_t will be less than t

           t = torch.tensor([timesteps[i]], device=image.device, dtype=torch.long)
           prev_t = torch.tensor([timesteps[i+1]], device=image.device, dtype=torch.long)

           # Ensure alphas_cumprod is on the same device
           #alphas_cumprod = stage_1.scheduler.alphas_cumprod.to(image.device)

           # TODO:
           # get alpha_bar and alpha_bar_prev for timestep t from alphas_cumprod
           # compute alpha

           alpha_bar = stage_1.scheduler.alphas_cumprod[t.item()]
           alpha_bar_prev = stage_1.scheduler.alphas_cumprod[prev_t.item()]
           alpha = alpha_bar / alpha_bar_prev

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

           # Get noise estimate
           model_output = stage_1.unet(
               image.half(),
               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], dim=1)

           # Scale the noise estimate to prevent extreme values
           noise_est = torch.clamp(noise_est, -3.0, 3.0)
```

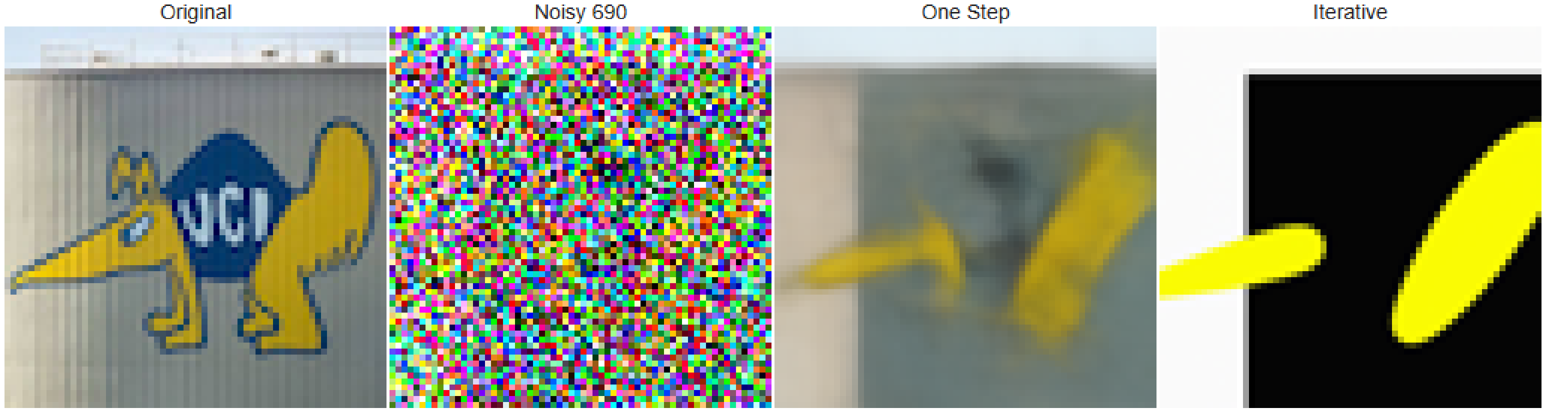
```
        # ===== your code here! =====

        x_0 = (image - (1 - alpha_bar).sqrt() * noise_est) / alpha_bar.sqrt() #predict x_0 from the noise_est and x_t
        pred_prev_image = (image - (1 - alpha_bar).sqrt() * noise_est) / alpha_bar.sqrt()

        print("Noise Est Shape:", noise_est.shape)
        print("Predicted Variance Shape:", predicted_variance.shape)
         # Add variance
        pred_prev_image = add_variance(predicted_variance, prev_t, pred_prev_image)

        # Clip the predicted previous image to prevent extreme values
        pred_prev_image = torch.clamp(pred_prev_image, -1.0, 1.0)

        # display intermeidate results. Remember that you will need to use .cpu()
        # to move tensors back to the cpu for display and permute/scale them
        if display and (i - i_start) % 5 == 0:
                  disp = {}
                  disp[f"Timestep {t}"] = pred_prev_image[0].cpu().permute(1, 2, 0) / 2. + 0.5
                  media.show_images(disp, width=256, height=256)


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

        image = pred_prev_image

    clean = image.cpu().detach()

    return clean
```

```
## run your experiments

# please use this prompt embedding for all experiments
prompt_embeds = prompt_embeds_dict["a high quality photo"]

# make it to 16
prompt_embeds = prompt_embeds.half()

# Load image
image_path = "zot.jpg"  # Ensure this file is uploaded to Colab
image = Image.open(image_path).convert("RGB")

# Transform the image to match the expected tensor format
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize to match the model input size
    transforms.ToTensor(),  # Convert to tensor (C, H, W) and normalize to [0,1]
    transforms.Normalize(0.5, 0.5)  # Normalize to [-1,1] (expected range for diffusion models)
])

test_im = transform(image).unsqueeze(0).to(device)
test_im = test_im.half()


# Add noise
i_start = 10
#t = strided_timesteps[i_start]

# Add noise to test image
t = torch.tensor([strided_timesteps[i_start]], device=test_im.device)

# Prepare the test image

im_noisy =  stage_1.scheduler.add_noise(test_im, noise=torch.randn_like(test_im), timesteps=t)

im_noisy = im_noisy.half()  # Ensure float16

# Iterative denoise
clean = iterative_denoise(im_noisy.to(device),i_start=i_start,prompt_embeds=prompt_embeds,timesteps=strided_timesteps)
```

```python
# One step denoise... as implemented previously
# ===== your code here! =====
# One-step denoising
model_output = stage_1.unet(im_noisy, t, encoder_hidden_states=prompt_embeds, return_dict=False)[0]

noise_est, _ = torch.split(model_output, im_noisy.shape[1], dim=1)
clean_one_step = (im_noisy - torch.sqrt(1 - stage_1.scheduler.alphas_cumprod[t].view(-1, 1, 1, 1)) * noise_est) / torch.sqrt(stage_1.scheduler.alphas_cumprod[t].view(-1, 1, 1, 1))
clean_one_step = clean_one_step.detach()

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



disp = {
    "Original": test_im[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
    f"Noisy {t.item()}": im_noisy[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
    "One Step": clean_one_step[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
    "Iterative": clean[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
}
media.show_images(disp, width=256, height=256)
```

```
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising:  87%|████████▋ | 20/23 [00:01<00:00, 12.41it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```


```
iterative denoising:  96%|█████████▌| 22/23 [00:01<00:00, 12.31it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising: 100%|██████████| 23/23 [00:01<00:00, 11.81it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```



2.2 Image generation by denoising pure noise [5pts]

In part 2, we used 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. Give this a try and show 5 results of `"a high quality photo"`.

Deliverables

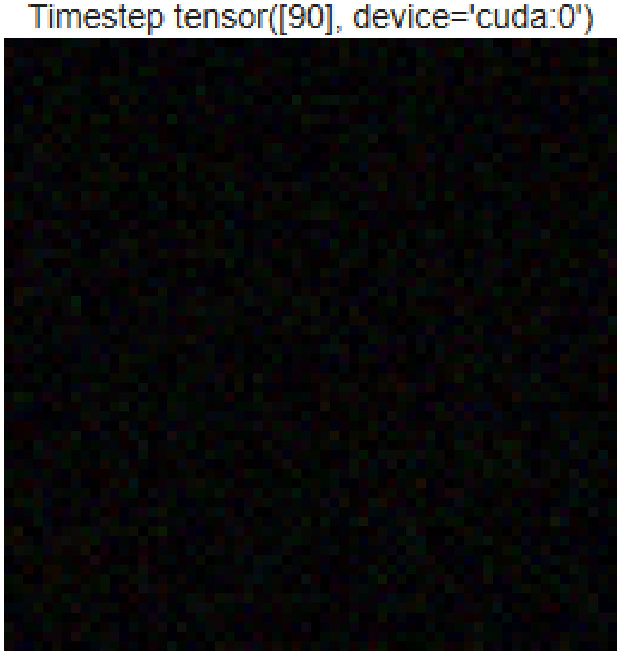
- Show 5 sampled images

Hints

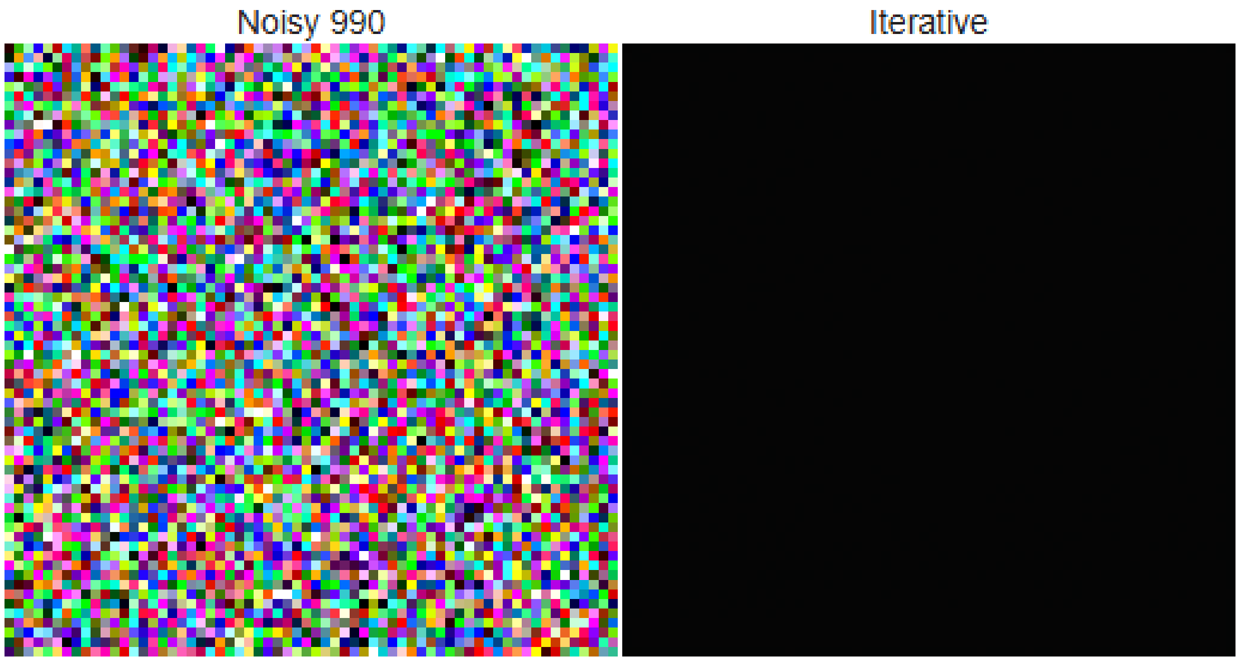
- Use `torch.randn` to make the noise.
- Make sure you move the tensor to the correct device and correct data type by calling `.half()` and `.to(device)`.
- The quality of the images will not be spectacular, but should be reasonable images. We will improve on this in the next step.

[17]
```
# Define the prompt embeddings
prompt_embeds = prompt_embeds_dict["a high quality photo"].to(device)

# Load and preprocess image
image_path = "zot.jpg"
image = Image.open(image_path).convert("RGB")
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize(0.5, 0.5)
])
test_im = transform(image).unsqueeze(0).to(device).half()

# Start denoising from pure noise
i_start = 0
t = strided_timesteps[i_start]

for i in range(5):
    im_noisy = torch.randn_like(test_im).to(device).half()

    # Denoise the image
    gen = iterative_denoise(im_noisy, i_start=i_start, prompt_embeds=prompt_embeds, timesteps=strided_timesteps)

    # Display the images
    disp = {
        f"Noisy {t}": im_noisy[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
        f"Iterative": gen[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5
    }
    media.show_images(disp, width=256, height=256)
```

```
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising:  91%|█████████ | 30/33 [00:02<00:00, 12.81it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```




```
iterative denoising:  97%|█████████▋| 32/33 [00:02<00:00, 12.68it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising: 100%|██████████| 33/33 [00:02<00:00, 12.34it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```

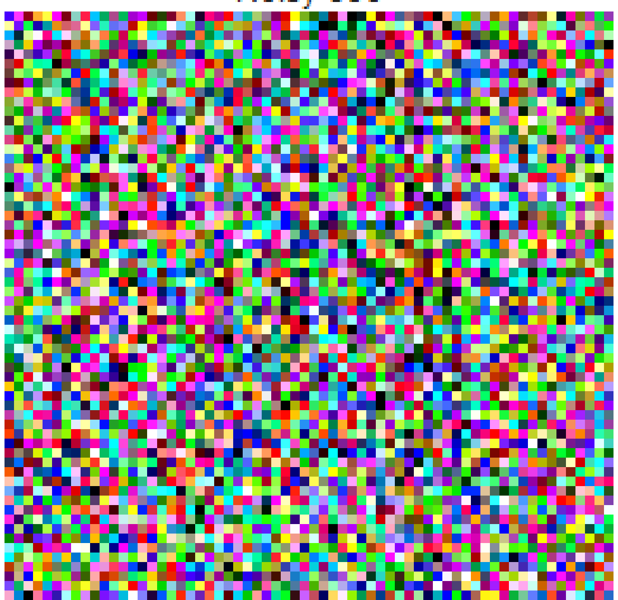


```
iterative denoising:  79%|████████▌  | 26/33 [00:02<00:00, 12.66it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising:  85%|████████▍  | 28/33 [00:02<00:00, 12.75it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising:  91%|█████████ | 30/33 [00:02<00:00, 12.80it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```

Timestep tensor([90], device='cuda:0')


```
iterative denoising:  97%|█████████▋| 32/33 [00:02<00:00, 12.72it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising: 100%|██████████| 33/33 [00:02<00:00, 12.77it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```

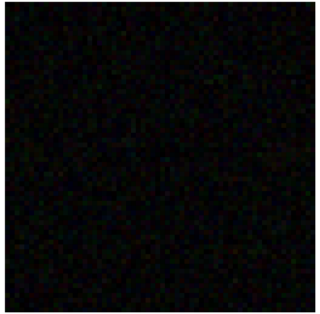
Noisy 990



Iterative

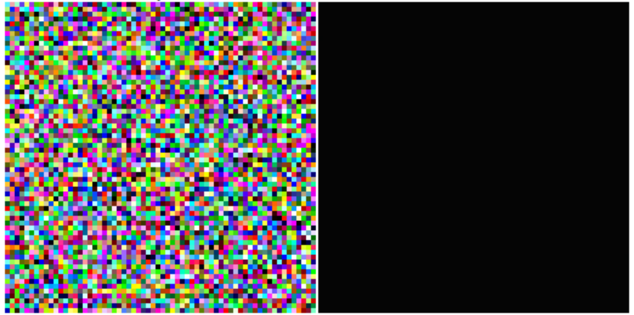
```
    media.show_images(disp, width=256, height=256)
Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```

Timestep tensor([90], device='cuda:0')



```
iterative denoising:  97%|█████████▋| 32/33 [00:02<00:00, 12.64it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising: 100%|██████████| 33/33 [00:02<00:00, 12.69it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```

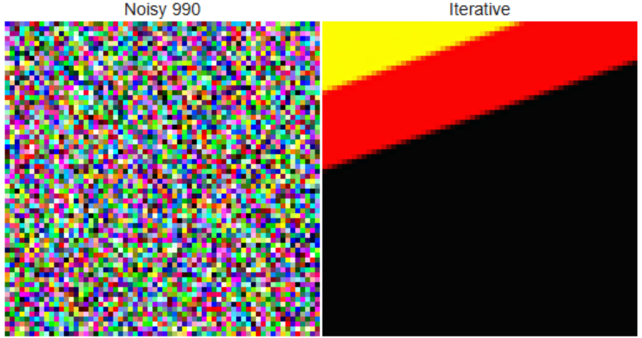
Noisy 990 Iterative



```
iterative denoising:   0%|          | 0/33 [00:00<?, ?it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
```

Timestep tensor([90], device='cuda:0')

```
iterative denoising:  97%|█████████▋| 32/33 [00:02<00:00, 12.26it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
iterative denoising: 100%|██████████| 33/33 [00:02<00:00, 11.89it/s]Noise Est Shape: torch.Size([1, 3, 64, 64])
Predicted Variance Shape: torch.Size([1, 3, 64, 64])
```



3. Classifier Free Guidance [10 pts]

You may have noticed that some of the generated images in the prior section are not great. 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

$$\epsilon = \epsilon_u + \gamma(\epsilon_c - \epsilon_u)$$

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 deault conditional guidance.

Deliverables

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

Hints

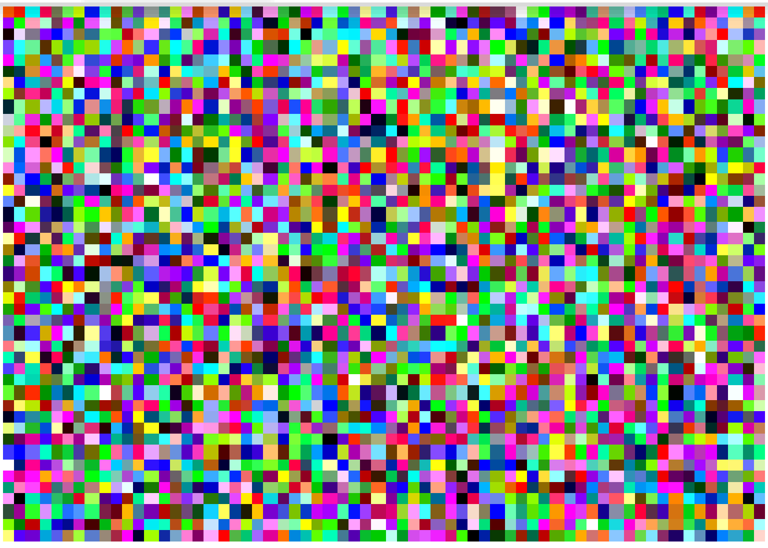
- You will need to run the UNet twice, once for the conditional prompt embedding, and once for the unconditional
- The UNet will predict both a conditional and an unconditional variance. Just use the conditional variance with the `add_variance` function.
- The resulting images should be more "photographic"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

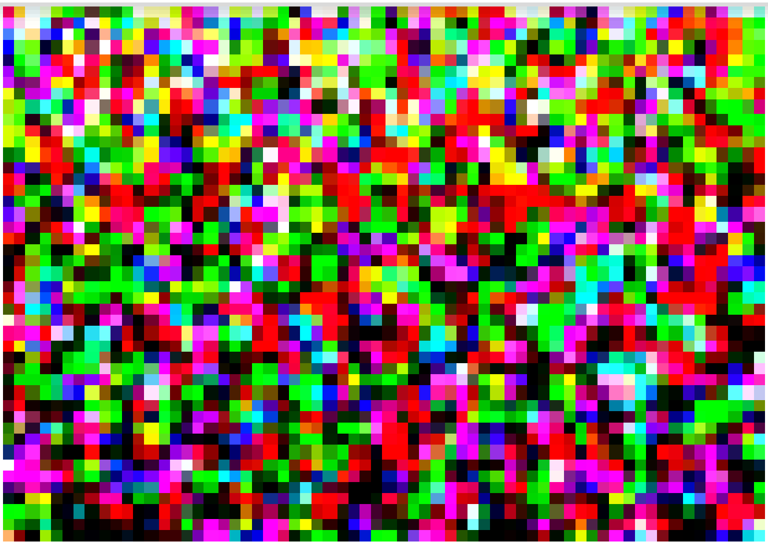
```
[45] def add_variance_task3(predicted_variance, t, image, timesteps):
        """
        Args:
            predicted_variance: (1, 3, 64, 64) tensor, last three channels of the UNet output
            t: scale tensor indicating timestep
            image: (1, 3, 64, 64) tensor, noisy image
            timesteps: a tensor of timesteps, similar to `strided_timesteps`

        Returns:
            (1, 3, 64, 64) tensor, image with the correct amount of variance added
        """
        # Ensure t is on the correct device and in long for indexing
        t = t.to(image.device).long()

        # Find the index of current timestep `t` in the full `timesteps` tensor
        index = (timesteps == t).nonzero(as_tuple=True)[0]
        if index.numel() == 0:  # Handle cases where t is not in timesteps
            index = torch.tensor(0, device=image.device)
        else:
            index = index[0]  # Take the first match

        # Retrieve the appropriate variance term
        variance = stage_1.scheduler._get_variance(
            timesteps[index].unsqueeze(0).to(image.device).long(),  # Ensure correct shape, device, and dtype
            predicted_variance=predicted_variance
        )

        variance_noise = torch.randn_like(image).half()  # Ensure noise is in float16

        # Scale down the variance addition
        variance = torch.exp(0.5 * variance) * variance_noise * 0.75

        # Clip the variance to prevent extreme values
        variance = torch.clamp(variance, -0.75, 0.75)

        return image + variance
```

```python
[46] def iterative_denoise_cfg(image, i_start, prompt_embeds, uncond_prompt_embeds, timesteps, gamma=7.0, vis=True):
        """
        Denoise an image iteratively using Classifier-Free Guidance (CFG).

        Args:
            image: (1, 3, 64, 64) tensor, noisy image
            i_start: int, starting index in the timesteps
            prompt_embeds: conditional prompt embedding
            uncond_prompt_embeds: unconditional (empty) prompt embedding
            timesteps: list of timesteps for denoising
            gamma: float, CFG scale factor
            vis: bool, whether to visualize intermediate results

        Returns:
            (1, 3, 64, 64) tensor, denoised image
        """
        with torch.no_grad():
            # Ensure image is in float16
            image = image.half()

            # Move timesteps to the same device as the image
            timesteps = torch.tensor(timesteps, device=image.device)

            # Move the scheduler's timesteps to the same device
            stage_1.scheduler.timesteps = stage_1.scheduler.timesteps.to(image.device)

            for i in tqdm(range(i_start, len(timesteps) - 1), "denoising"):
                if i >= len(timesteps) - 1:  # Ensure index is in range
                    break

                # Get current and previous timesteps
                t = timesteps[i].long()  # Current timestep
                prev_t = timesteps[i + 1].long()  # Previous timestep

                # Get alpha_bar values for current and previous timesteps
                alpha_bar = stage_1.scheduler.alphas_cumprod[t].to(image.device)
                alpha_bar_prev = stage_1.scheduler.alphas_cumprod[prev_t].to(image.device)

                # Ensure prompt embeddings are in float16
                prompt_embeds = prompt_embeds.half()
                uncond_prompt_embeds = uncond_prompt_embeds.half()
```

```
        #-----------

        # Ensure t is in float16 for the model
        t_float16 = t.float().half()

        # Get conditional noise estimate
        cond_output = stage_1.unet(
            image.half(),
            t_float16,
            encoder_hidden_states=prompt_embeds.to(image.device),
            return_dict=False
        )[0]

        # Get unconditional noise estimate
        uncond_output = stage_1.unet(
            image.half(),
            t_float16,
            encoder_hidden_states=uncond_prompt_embeds.to(image.device),
            return_dict=False
        )[0]

        # Split outputs into noise estimates and predicted variances
        cond_noise_est, cond_variance = torch.split(cond_output, image.shape[1], dim=1)
        uncond_noise_est, _ = torch.split(uncond_output, image.shape[1], dim=1)

        # Combine noise estimates using CFG formula
        noise_est = uncond_noise_est + gamma * (cond_noise_est - uncond_noise_est)

        # Predict x_0 (clean image) from the noise estimate and x_t
        x_t = image
        x_0 = (x_t - torch.sqrt(1 - alpha_bar) * noise_est) / torch.sqrt(alpha_bar)

        # Predict the previous image (x_{t-1})
        pred_prev_image = torch.sqrt(alpha_bar_prev) * x_0 + torch.sqrt(1 - alpha_bar_prev) * noise_est

        # Add variance to the predicted image
        pred_prev_image = add_variance_task3(cond_variance, prev_t, pred_prev_image, timesteps)

        # Update the image for the next iteration
        image = pred_prev_image

    # Return the final denoised image
    clean = image.cpu().detach()
    return clean
```

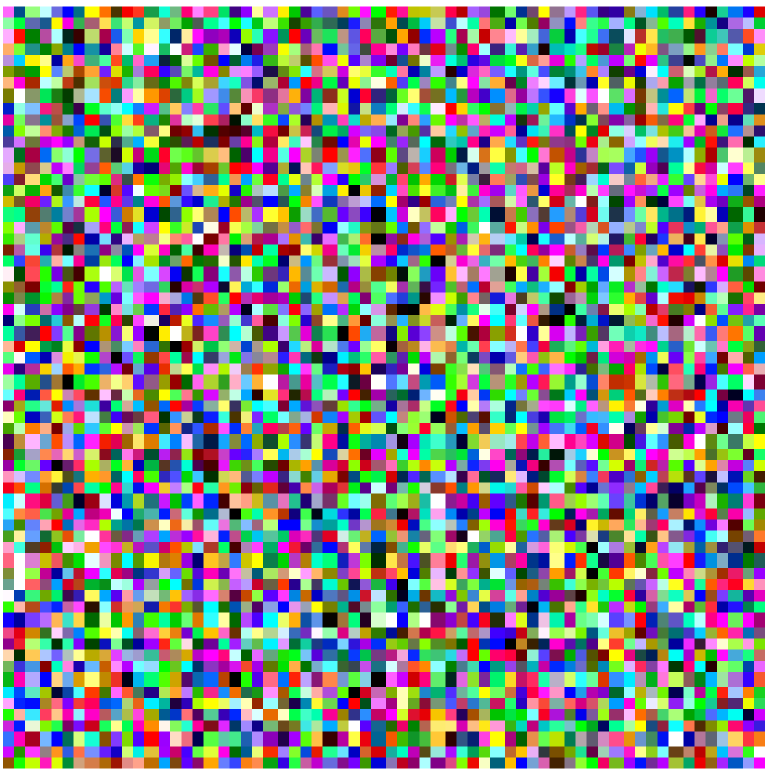
```
[47] # The condition prompt embedding
     prompt_embeds = prompt_embeds_dict['a high quality photo'].half()  # Ensure float16
     # The unconditional prompt embedding
     uncond_prompt_embeds = prompt_embeds_dict[''].half()  # Ensure float16

     # Add noise
     i_start = 0
     t = strided_timesteps[i_start]

     for i in range(5):
         # Generate noisy image
         image_tensor = torchvision.transforms.ToTensor()(image).unsqueeze(0).to(device)  # Convert image to tensor
         im_noisy = torch.randn_like(image_tensor).half()  # Generate Gaussian noise in float16

         # Denoise using CFG
         gen = iterative_denoise_cfg(im_noisy.to(device),
                                     i_start=i_start,
                                     prompt_embeds=prompt_embeds.to(device),  # Ensure prompt_embeds is in float16
                                     uncond_prompt_embeds=uncond_prompt_embeds.to(device),  # Ensure uncond_prompt_embeds is in float16
                                     timesteps=strided_timesteps,
                                     gamma=7.0)

         # Display the images
         disp = {
             f"Noisy {t}": im_noisy[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5,
             f"Iterative": gen[0].cpu().permute(1, 2, 0).numpy() / 2. + 0.5
         }
         media.show_images(disp, width=256, height=256)
```

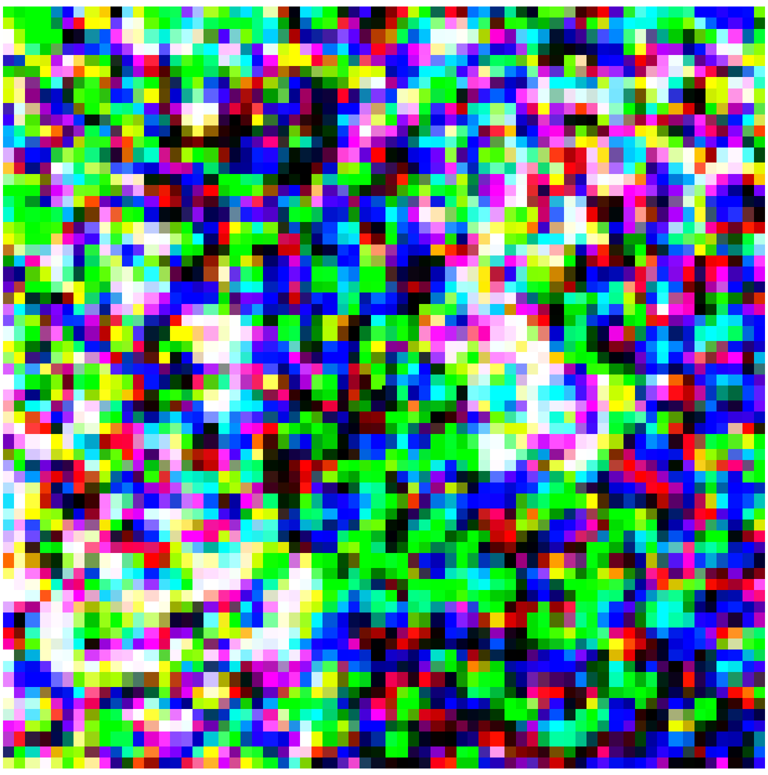




denoising: 100%|██████████| 33/33 [00:05<00:00, 6.34it/s]

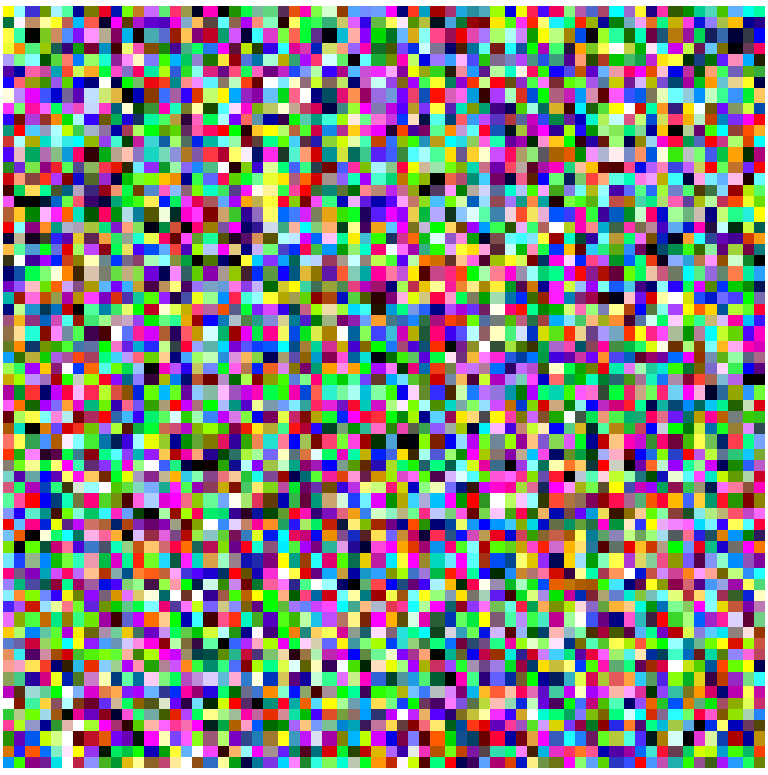
Noisy 990 Iterative

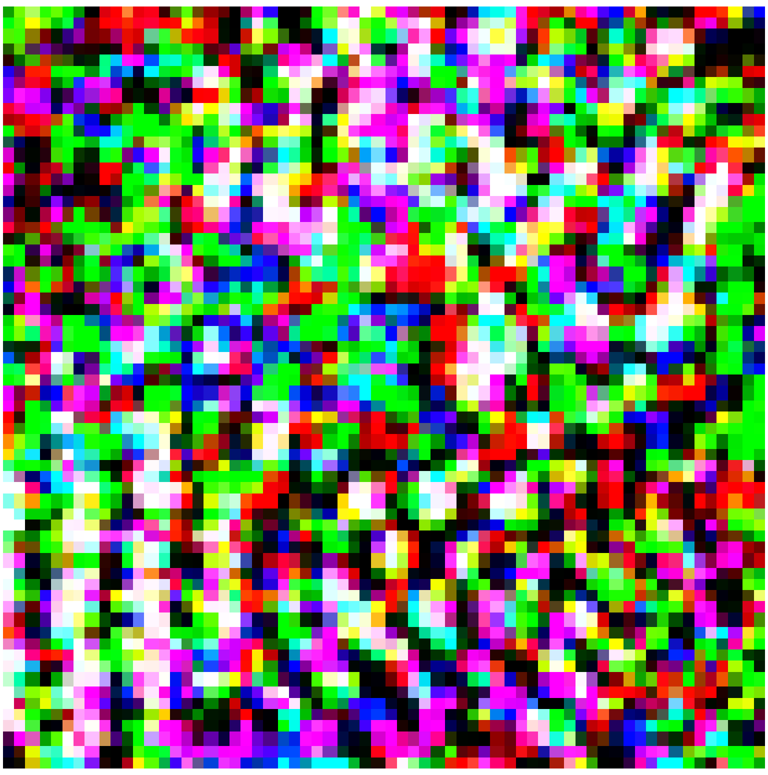




denoising: 100%|██████████| 33/33 [00:05<00:00, 6.20it/s]

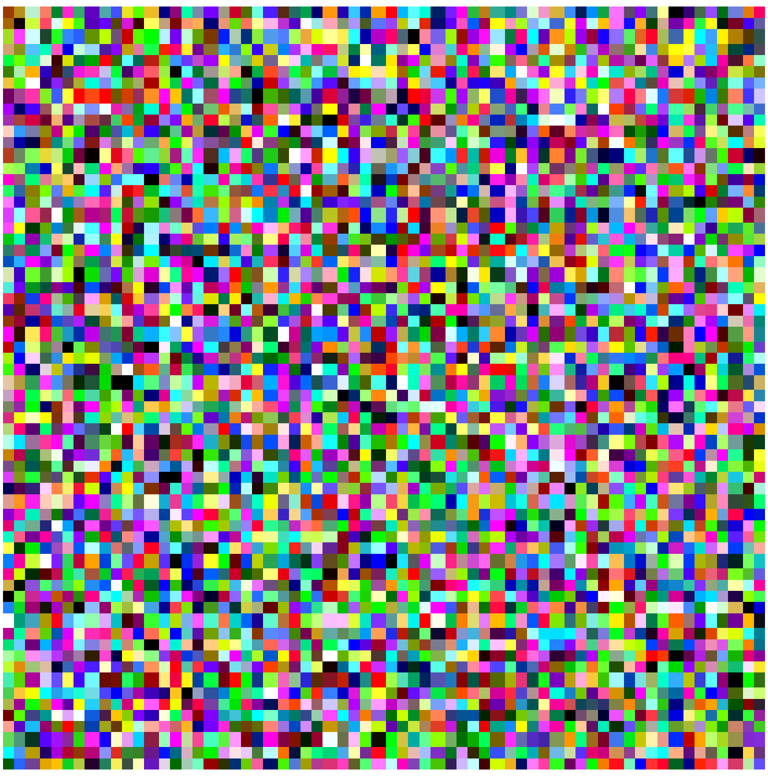
Noisy 990 Iterative

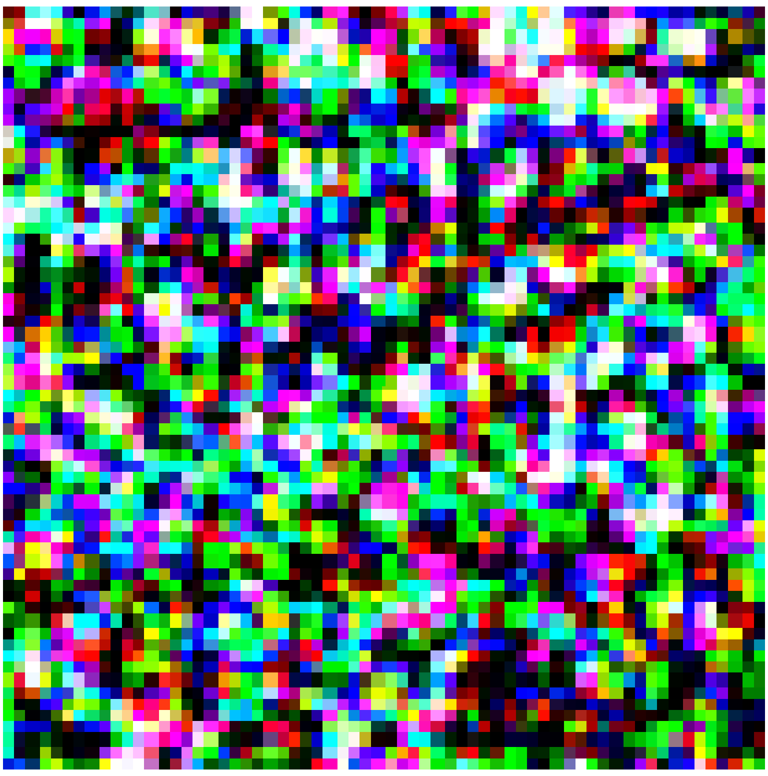




denoising: 100%|██████████| 33/33 [00:05<00:00, 6.19it/s]

Noisy 990 Iterative





4. Image-to-image Translation

![image.png]

Now we will experiment with taking a real image, add noise to it, and then denoise. This effectively allows us to make modiications 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 but we won't necessarily end up where we started.

To start, 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.

Please run the forward process to get a noisy test image, and then run your `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 see a range of images that start randomly and gradually look more like the original image
- You may find the provided utility functions useful for grabbing a local image or an image off the web and converting it appropriate for input to the model

4.1 Implementation [10pts]

[51]
```
import torch
import torchvision.transforms as transforms
from PIL import Image, UnidentifiedImageError
import requests
from io import BytesIO
import mediapy as media
```

[52]
```
def image2image(image,ivals = [1,3,5,7,10,20],vis=True):
    # Please use this prompt, as an "unconditional" text prompt
    prompt_embeds = prompt_embeds_dict["a high quality photo"]
    uncond_prompt_embeds = prompt_embeds_dict['']

    img_in={}
    img_out={}
    for i_start in ivals:
        t = strided_timesteps[i_start]

        noisy = iterative_denoise_cfg(image.half().to(device), prompt_embeds, uncond_prompt_embeds, t)

        gen = iterative_denoise_cfg(noisy.half().to(device), prompt_embeds, uncond_prompt_embeds, t)

        img_in[i_start] = noisy[0].permute(1,2,0)/2.+0.5
        img_out[i_start] = gen[0].permute(1,2,0)/2.+0.5

    if vis:
        media.show_images(img_in,width=256,height=256,columns=6)
        media.show_images(img_out,width=256,height=256,columns=6)

    return img_out
```

[53]
```python
#
# utility code for converting an image into an appropriately scaled tensor
#
def process_pil_im(img,vis=True):
    '''
    Transform a PIL image into a tensor of size [1,3,64,64]
    '''

    # 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
    if vis:
        print("Processed image")
        media.show_image(img[0].permute(1,2,0) / 2 + 0.5,height=256,width=256)

    return img
```

[54]
```
################
# load an image from a given URL
### (please use your own URL below"
################
url = "https://parkerlab.bio.uci.edu/pictures/photography%20pictures/2020_10_20_UCI_buildings/_5R_9295_tweak.jpg"
response = requests.get(url)
web_im = Image.open(BytesIO(response.content))
# The function below crops out the center of the image if it is not square.
# you may want to manually crop before that step, e.g. web_im = web_im.crop((3000,500,7016,4872))
web_im = process_pil_im(web_im)
```

Processed image

```python
[57] def image2image(image, ivals=[1, 3, 5, 7, 10, 20], vis=True):
        """
        Perform image-to-image editing using Classifier-Free Guidance (CFG).
        """
        # Use this prompt as an "unconditional" text prompt
        prompt_embeds = prompt_embeds_dict["a high quality photo"]
        uncond_prompt_embeds = prompt_embeds_dict['']

        img_in = {}
        img_out = {}
        for i_start in ivals:
            t = strided_timesteps[i_start]

            # Add noise to the image
            noisy = iterative_denoise_cfg(image.half().to(device), i_start, prompt_embeds, uncond_prompt_embeds, strided_timesteps)

            # Denoise the noisy image
            gen = iterative_denoise_cfg(noisy.half().to(device), i_start, prompt_embeds, uncond_prompt_embeds, strided_timesteps)

            # Store the noisy and denoised images
            img_in[i_start] = noisy[0].permute(1, 2, 0) / 2. + 0.5
            img_out[i_start] = gen[0].permute(1, 2, 0) / 2. + 0.5

        # Visualize the results
        if vis:
            media.show_images(img_in, width=256, height=256, columns=6)
            media.show_images(img_out, width=256, height=256, columns=6)

        return img_out

    # Load an image from a URL
    url = "https://upload.wikimedia.org/wikipedia/commons/thumb/1/18/Lewis_Hamilton_2016_Malaysia_2.jpg/800px-Lewis_Hamilton_2016_Malaysia_2.jpg"
    try:
        # Send a GET request to the URL
        response = requests.get(url)
        response.raise_for_status()  # Raise an error for bad status codes (e.g., 404, 500)

        # Check if the response contains valid image data
        if 'image' not in response.headers.get('Content-Type', ''):
            raise ValueError("The URL does not point to a valid image file.")

        # Open the image using PIL
        my_im = Image.open(BytesIO(response.content))
        print("Image successfully loaded from URL.")

        # Process the image
        my_im = process_pil_im(my_im)

        # Perform image-to-image editing
        image2image(my_im, ivals=[1, 3, 5, 7, 10, 20])
```

```
except requests.exceptions.RequestException as e:
    print(f"Failed to fetch the image from the URL: {e}")
except UnidentifiedImageError:
    print("The downloaded file is not a valid image.")
except Exception as e:
    print(f"An error occurred: {e}")
```

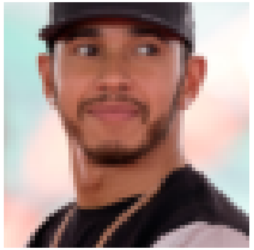
```
Failed to fetch the image from the URL: 403 Client Error: Forbidden. Please comply with the User-Agent policy: https://meta.wikimedia.org/wiki/User-Agent_policy for url: https://upload.wikimedia.org/wikipedia/commons/thumb/1/18/Lewis_Hamilton_2016_Malaysia_2.jpg/800px-Lewis_Hamilton_2016_Malaysi
```

```
[ ] # test with test_im and two other images of your own choosing
image2image(test_im);
image2image(web_im);
.
.
etc
.
.
```

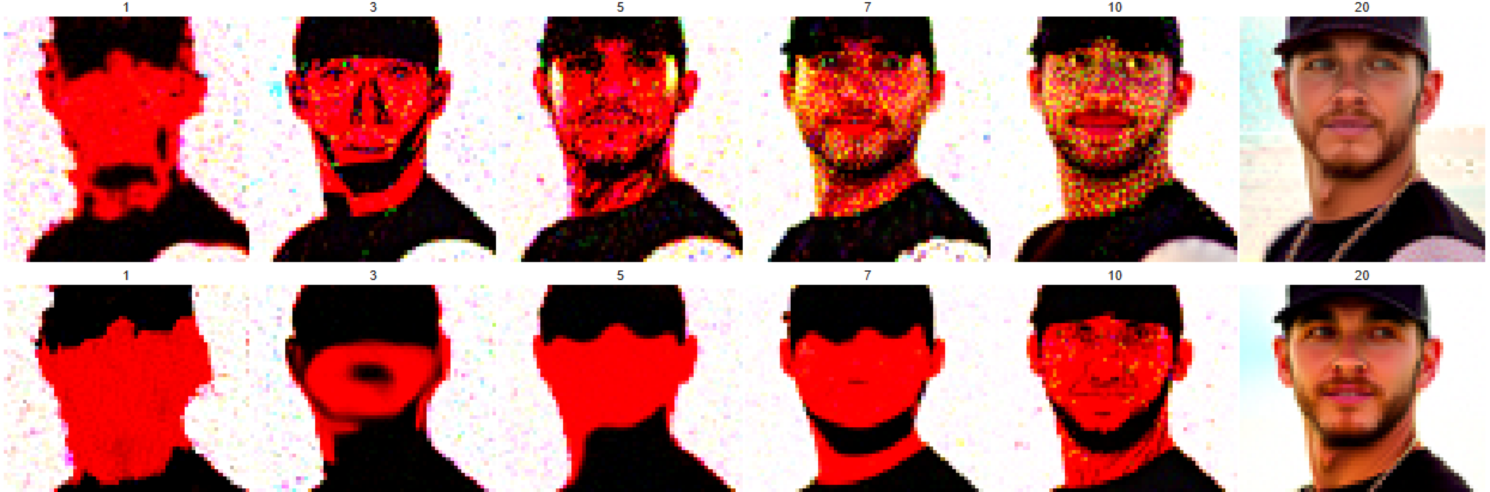
```
import torch
import torchvision.transforms as transforms
from PIL import Image, UnidentifiedImageError
import requests
from io import BytesIO
import mediapy as media

# Utility function to process a PIL image into a tensor
def process_pil_im(img, vis=True):
    """
    Transform a PIL image into a tensor of size [1, 3, 64, 64].
    """
    # 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
    if vis:
        print("Processed image")
        media.show_image(img[0].permute(1, 2, 0) / 2 + 0.5, height=256, width=256)

    return img

# Function to perform image-to-image editing using CFG
def image2image(image, ivals=[1, 3, 5, 7, 10, 20], vis=True):
    """
    Perform image-to-image editing using Classifier-Free Guidance (CFG).
    """
    # Use this prompt as an "unconditional" text prompt
    prompt_embeds = prompt_embeds_dict["a high quality photo"]
    uncond_prompt_embeds = prompt_embeds_dict['']

    img_in = {}
    img_out = {}
    for i_start in ivals:
        t = strided_timesteps[i_start]

        # Add noise to the image
        noisy = iterative_denoise_cfg(image.half().to(device), i_start, prompt_embeds, uncond_prompt_embeds, strided_timesteps)

        # Denoise the noisy image
        gen = iterative_denoise_cfg(noisy.half().to(device), i_start, prompt_embeds, uncond_prompt_embeds, strided_timesteps)
```

```python
        # Store the noisy and denoised images
        img_in[i_start] = noisy[0].permute(1, 2, 0) / 2. + 0.5
        img_out[i_start] = gen[0].permute(1, 2, 0) / 2. + 0.5

    # Visualize the results
    if vis:
        media.show_images(img_in, width=256, height=256, columns=6)
        media.show_images(img_out, width=256, height=256, columns=6)

    return img_out

# Load an image from a URL
url = "https://upload.wikimedia.org/wikipedia/commons/thumb/1/18/Lewis_Hamilton_2016_Malaysia_2.jpg/800px-Lewis_Hamilton_2016_Malaysia_2.jpg"
try:
    # Send a GET request to the URL with a valid User-Agent header
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"
    }
    response = requests.get(url, headers=headers)
    response.raise_for_status()  # Raise an error for bad status codes (e.g., 404, 500)

    # Check if the response contains valid image data
    if 'image' not in response.headers.get('Content-Type', ''):
        raise ValueError("The URL does not point to a valid image file.")

    # Open the image using PIL
    my_im = Image.open(BytesIO(response.content))
    print("Image successfully loaded from URL.")

    # Process the image
    my_im = process_pil_im(my_im)

    # Perform image-to-image editing
    image2image(my_im, ivals=[1, 3, 5, 7, 10, 20])

except requests.exceptions.RequestException as e:
    print(f"Failed to fetch the image from the URL: {e}")
except UnidentifiedImageError:
    print("The downloaded file is not a valid image.")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Image successfully loaded from URL.
Processed image
```



```
<ipython-input-46-ac8be00ebc51>:22: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  timesteps = torch.tensor(timesteps, device=image.device)
denoising: 100%|██████████| 32/32 [00:04<00:00,  6.43it/s]
denoising: 100%|██████████| 32/32 [00:04<00:00,  6.41it/s]
denoising: 100%|██████████| 30/30 [00:04<00:00,  6.44it/s]
denoising: 100%|██████████| 30/30 [00:04<00:00,  6.34it/s]
denoising: 100%|██████████| 28/28 [00:04<00:00,  6.34it/s]
denoising: 100%|██████████| 28/28 [00:04<00:00,  6.30it/s]
denoising: 100%|██████████| 26/26 [00:04<00:00,  6.19it/s]
denoising: 100%|██████████| 26/26 [00:04<00:00,  6.24it/s]
denoising: 100%|██████████| 23/23 [00:03<00:00,  6.22it/s]
denoising: 100%|██████████| 23/23 [00:03<00:00,  6.15it/s]
denoising: 100%|██████████| 13/13 [00:02<00:00,  6.19it/s]
denoising: 100%|██████████| 13/13 [00:02<00:00,  6.17it/s]
```



4.2 Sketch-to-Image [5pts]

This procedure is particularly fun 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 where they are mapped back onto the natural image manifold.

For this part you can experiment with images from the web but at least one input should be an image that you drew yourself (e.g. using some paint program or even an image editor on your phone). 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)
- 1 hand drawn image, edited using the above method for noise levels [1, 3, 5, 7, 10, 20] (and whatever additional noise levels you want)

Hints

- Use the preprocessing code above to convert web images to the format expected by DeepFloyd.

5. Inpainting [15 pts]

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.:

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

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

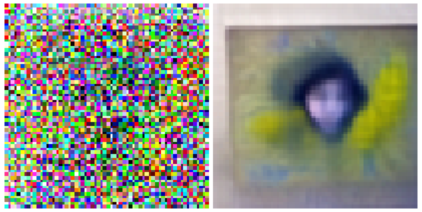
[28]
```python
def add_variance_task3(predicted_variance, t, image, timesteps):
    """
    Args:
        predicted_variance: (1, 3, 64, 64) tensor, last three channels of the UNet output
        t: scale tensor indicating timestep
        image: (1, 3, 64, 64) tensor, noisy image
        timesteps: a tensor of timesteps, similar to `strided_timesteps`

    Returns:
        (1, 3, 64, 64) tensor, image with the correct amount of variance added
    """
    # Ensure t is on the correct device and in long for indexing
    t = t.to(image.device).long()

    # Ensure timesteps is on the correct device
    timesteps = timesteps.to(image.device)

    # Find the index of current timestep `t` in the full `timesteps` tensor
    index = (timesteps == t).nonzero(as_tuple=True)[0]
    if index.numel() == 0:  # Handle cases where t is not in timesteps
        index = torch.tensor(0, device=image.device)
    else:
        index = index[0]  # Take the first match

    # Retrieve the appropriate variance term
    variance = stage_1.scheduler._get_variance(
        timesteps[index].unsqueeze(0).to(image.device).long(),  # Ensure correct shape, device, and dtype
        predicted_variance=predicted_variance
    )

    variance_noise = torch.randn_like(image).half()  # Ensure noise is in float16

    # Scale down the variance addition
    variance = torch.exp(0.5 * variance) * variance_noise * 0.75

    # Clip the variance to prevent extreme values
    variance = torch.clamp(variance, -0.75, 0.75)

    return image + variance


def inpaint(original_image, mask, prompt_embeds, uncond_prompt_embeds, timesteps, istart=0, vis=True):
    original_image = original_image.to(device).half()
    image = forwardnoise(original_image, timesteps[istart]).to(device).half()
    mask = mask.to(device).half()

    # Ensure timesteps is a PyTorch tensor and on the correct device
    if not isinstance(timesteps, torch.Tensor):
        timesteps = torch.tensor(timesteps, device=device)
    else:
        timesteps = timesteps.to(device)

    # Ensure stage_1.scheduler.timesteps is on the correct device
    stage_1.scheduler.timesteps = stage_1.scheduler.timesteps.to(device)
```

```
with torch.no_grad():
    for i in tqdm(range(istart, len(timesteps) - 1), "denoising"):
        if i >= len(timesteps) - 1:  # Ensure index is in range
            break

        # Get timesteps and ensure they are on the correct device
        t = timesteps[i].to(device)  # Move to the correct device
        prev_t = timesteps[i + 1].to(device)  # Move to the correct device

        # Parameters
        gamma = 7
        alpha_bar = stage_1.scheduler.alphas_cumprod[t].to(device)  # Ensure alpha_bar is on the correct device
        alpha_bar_prev = stage_1.scheduler.alphas_cumprod[prev_t].to(device)  # Ensure alpha_bar_prev is on the correct device
        alpha = alpha_bar / alpha_bar_prev

        # Ensure prompt_embeds and uncond_prompt_embeds are in float16
        prompt_embeds = prompt_embeds.half()
        uncond_prompt_embeds = uncond_prompt_embeds.half()

        # Ensure t is in float16 for the model
        t_float16 = t.float().half()

        # Get noise estimate
        model_output = stage_1.unet(
            image.half(),
            t_float16,  # Ensure t is in float16
            encoder_hidden_states=prompt_embeds.to(device),  # Ensure prompt_embeds is on the correct device
            return_dict=False
        )[0]

        # Get uncond noise estimate
        uncond_model_output = stage_1.unet(
            image.half(),
            t_float16,  # Ensure t is in float16
            encoder_hidden_states=uncond_prompt_embeds.to(device),  # Ensure uncond_prompt_embeds is on the correct device
            return_dict=False
        )[0]

        # Split estimate into noise and variance estimate
        cond_noise_est, predicted_variance = torch.split(model_output, image.shape[1], dim=1)
        uncond_noise_est, _ = torch.split(uncond_model_output, image.shape[1], dim=1)
        noise_est = uncond_noise_est + gamma * (cond_noise_est - uncond_noise_est)  # Combination of conditional and unconditional

        # Predict x_0 from the noise_est and x_t
        x_t = image
        x_0 = (x_t - torch.sqrt(1 - alpha_bar) * noise_est) / torch.sqrt(alpha_bar)

        # Predict the previous image
```

```python
            # Predict the previous image #--------------
            pred_prev_image = torch.sqrt(alpha_bar_prev) * x_0 + torch.sqrt(1 - alpha_bar_prev) * noise_est

            # Add noise using the new add_variance_task3 function
            pred_prev_image = add_variance_task3(predicted_variance, prev_t, pred_prev_image, timesteps)

            # Mask the result and replace the background with a noisy version of the original_image
            noisy_original = forwardnoise(original_image, prev_t).to(device).half()
            image = mask * pred_prev_image + (1 - mask) * noisy_original

            # Show intermediate results every 5 iterations
            if vis and i % 5 == 0:
                x_0 = x_0.cpu().detach()[0]
                x_t = image.cpu().detach()[0]
                media.show_images({
                    'x_t': x_t.permute(1, 2, 0) / 2. + 0.5,
                    'x_0': x_0.permute(1, 2, 0) / 2. + 0.5
                }, width=128, height=128)

    return image.cpu().detach()

# Example usage
prompt_embeds = prompt_embeds_dict["a high quality photo"].half()
uncond_prompt_embeds = prompt_embeds_dict[''].half()

# Make a mask
mask = torch.zeros_like(test_im)
mask[:, :, 2:20, 16:42] = 1.0
mask = TF.gaussian_blur(mask, 9, 2)
mask = mask.to(device)

# Ensure timesteps is a PyTorch tensor and on the correct device
if not isinstance(strided_timesteps, torch.Tensor):
    strided_timesteps = torch.tensor(strided_timesteps, device=device)
else:
    strided_timesteps = strided_timesteps.to(device)

# Ensure stage_1.scheduler.timesteps is on the correct device
stage_1.scheduler.timesteps = stage_1.scheduler.timesteps.to(device)

# Inpaint
filled_im = inpaint(test_im, mask, prompt_embeds, uncond_prompt_embeds, strided_timesteps, istart=5, vis=True)

test_im_cpu = test_im.cpu()
mask_cpu = mask.cpu()
filled_im_cpu = filled_im.cpu()
```

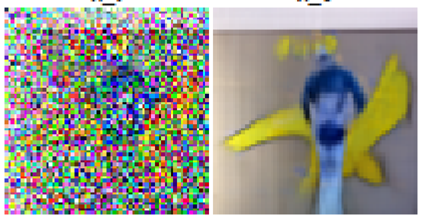
```
# Visualize mask
media.show_images({
    'Image': test_im_cpu[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Mask': mask_cpu[0].permute(1, 2, 0).numpy(),
    'To Replace': (test_im_cpu * mask_cpu)[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Result': filled_im_cpu[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Filled': (filled_im_cpu * mask_cpu)[0].permute(1, 2, 0).numpy() / 2. + 0.5,
}, height=256, width=256)
```

x_t x_0



```
denoising:  36%|███       | 10/28 [00:01<00:02,  6.36it/s]
```

x_t x_0



```
denoising:  54%|█████     | 15/28 [00:02<00:02,  6.38it/s]
```

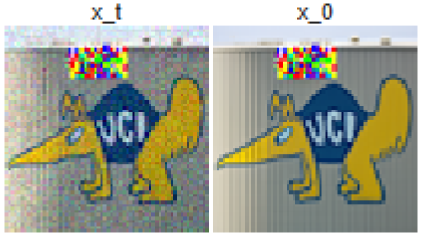
x_t x_0

```
denoising:  71%|███████   | 20/28 [00:03<00:01,  6.23it/s]
```

x_t x_0

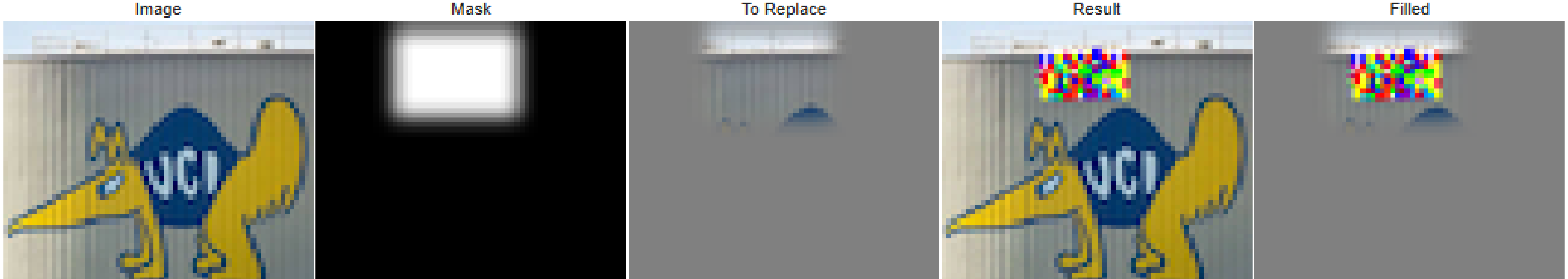
```
denoising:  89%|█████████ | 25/28 [00:04<00:00,  6.24it/s]
```

x_t x_0



```
denoising: 100%|██████████| 28/28 [00:04<00:00,  6.16it/s]
```

Image Mask To Replace Result Filled



6. Text-Conditioned Inpainting [10 pts]

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.

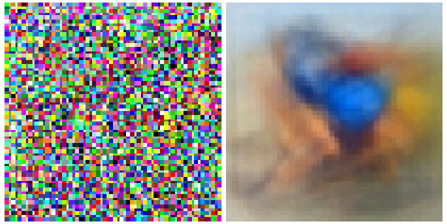
Hints

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

[29]
```python
# Define the prompt embeddings for text-conditioned inpainting
prompt_embeds = prompt_embeds_dict["a rocket ship"].half()  # Use the desired prompt
uncond_prompt_embeds = prompt_embeds_dict[''].half()  # Unconditional prompt

# Make a mask (example: mask the top-left corner of the image)
mask = torch.zeros_like(test_im)  # Create a mask with the same shape as the test image
mask[:, :, 2:20, 16:42] = 1.0  # Define the region to inpaint (e.g., top-left corner)
mask = TF.gaussian_blur(mask, 9, 2)  # Apply Gaussian blur to smooth the mask edges
mask = mask.to(device)  # Move the mask to the GPU

# Inpaint the image using the text-conditioned prompt
filled_im = inpaint(
    test_im,  # Original image
    mask,  # Mask defining the region to inpaint
    prompt_embeds,  # Text-conditioned prompt embeddings
    uncond_prompt_embeds,  # Unconditional prompt embeddings
    strided_timesteps,  # Timesteps for diffusion
    istart=5,  # Starting timestep
    vis=True  # Visualize intermediate results
)

# Move tensors to CPU for visualization
test_im_cpu = test_im.cpu()
mask_cpu = mask.cpu()
filled_im_cpu = filled_im.cpu()

# Visualize the results
media.show_images({
    'Image': test_im_cpu[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Mask': mask_cpu[0].permute(1, 2, 0).numpy(),
    'To Replace': (test_im_cpu * mask_cpu)[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Result': filled_im_cpu[0].permute(1, 2, 0).numpy() / 2. + 0.5,
    'Filled': (filled_im_cpu * mask_cpu)[0].permute(1, 2, 0).numpy() / 2. + 0.5,
}, height=256, width=256)

# Show upsampled result if desired
# final = upsample(filled_im, prompt_embeds)
# media.show_image(final[0].permute(1, 2, 0) / 2 + 0.5)
```

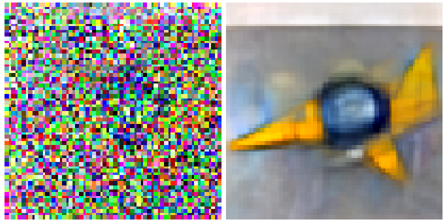
```
denoising:   0%|          | 0/28 [00:00<?, ?it/s]
```

x_t x_0



```
denoising:  18%|██        | 5/28 [00:00<00:03,  6.18it/s]
```

x_t x_0



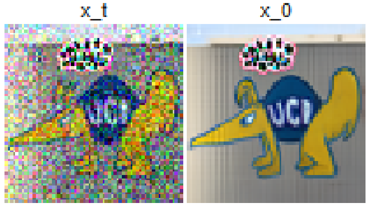
```
denoising:  36%|███       | 10/28 [00:01<00:02,  6.35it/s]
```

x_t x_0

```
denoising:  54%|█████     | 15/28 [00:02<00:02,  6.39it/s]
```

x_t x_0

```
denoising:  71%|███████   | 20/28 [00:03<00:01,  6.39it/s]
```




```
denoising:  89%|████████▉ | 25/28 [00:04<00:00,  6.37it/s]
```


```
denoising: 100%|██████████| 28/28 [00:04<00:00,  6.26it/s]
```