Project 5: Fun With Diffusion Models!

Part A: The Power of Diffusion Models!

Overview

In the first part of this project I create interesting images and optical illusions using diffusion models, implementing variations of diffusion sampling loops.

Part 0: Setup

We use DeepFloyd IF for all the images generated in this part of the project, accessed via Hugging Face. It consists of 2 stages, the first of which produces images of size 64 x 64, and the second which takes those outputs and generates images of size 256 x 256.

I upsample all of the generated images in the second half of part A to 256 x 256 if they are not already 256 x 256, and use a random seed of **180** for all of the proceeding generations.

Below are the generated images for 3 prompts to the model with num_inference_steps = 5 and num_inference_steps = 20 for both stages.

num_inference_steps=5

an oil painting of a snowy mountain village

a man wearing a hat

a rocket ship

num_inference_steps=20

an oil painting of a snowy mountain village

a man wearing a hat

a rocket ship

The images match the text prompts pretty well, even for a small number of inference steps, however the quality of the output varies based on the number of inference steps. There are visible noise artifacts from the generations with a small number of inference steps due to the upscaling (second stage) not being run for enough iterations. In addition, the images generated do not look very realistic (the proportions of the man's face are off as well as the number of fingers on his hand, as well as the rocket's flames being inconsistently drawn) which is probably due to the first stage not being run for enough inference steps. Larger inference steps seem to fix this, however the resulting images seem more cartoonish.

Part 1: Sampling Loops

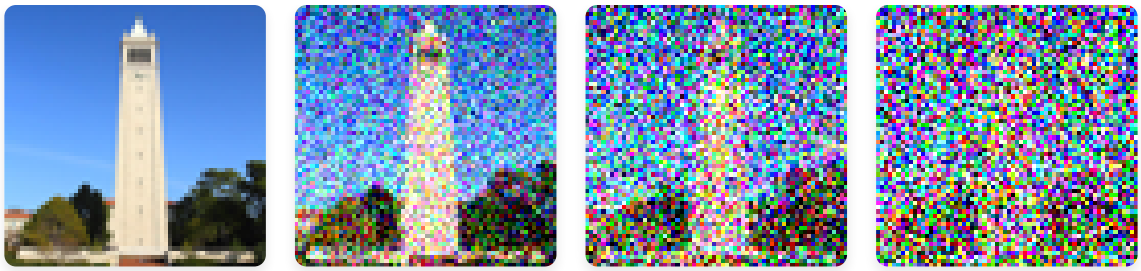
We can start with a clean image x0 and iteratively add noise sampled from a Gaussian distribution for each timestep t until we end up with pure noise at timestep T. A diffusion model learns to reverse this process, given a noisy xt and timestep t it predicts the noise in the image. With this we can estimate the initial x0 or remove part of the noise, giving us xt-1. For DeepFloyd models, T=1000, and there are particular noise coefficients alpha bar t that dictate how we should add noise.

1.1 Implementing the Forward Process

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

In the forward process, we add noise to an image following the above equation for t from 0 to 999. This gives us progressively more noisy images, as shown below (not upscaled).

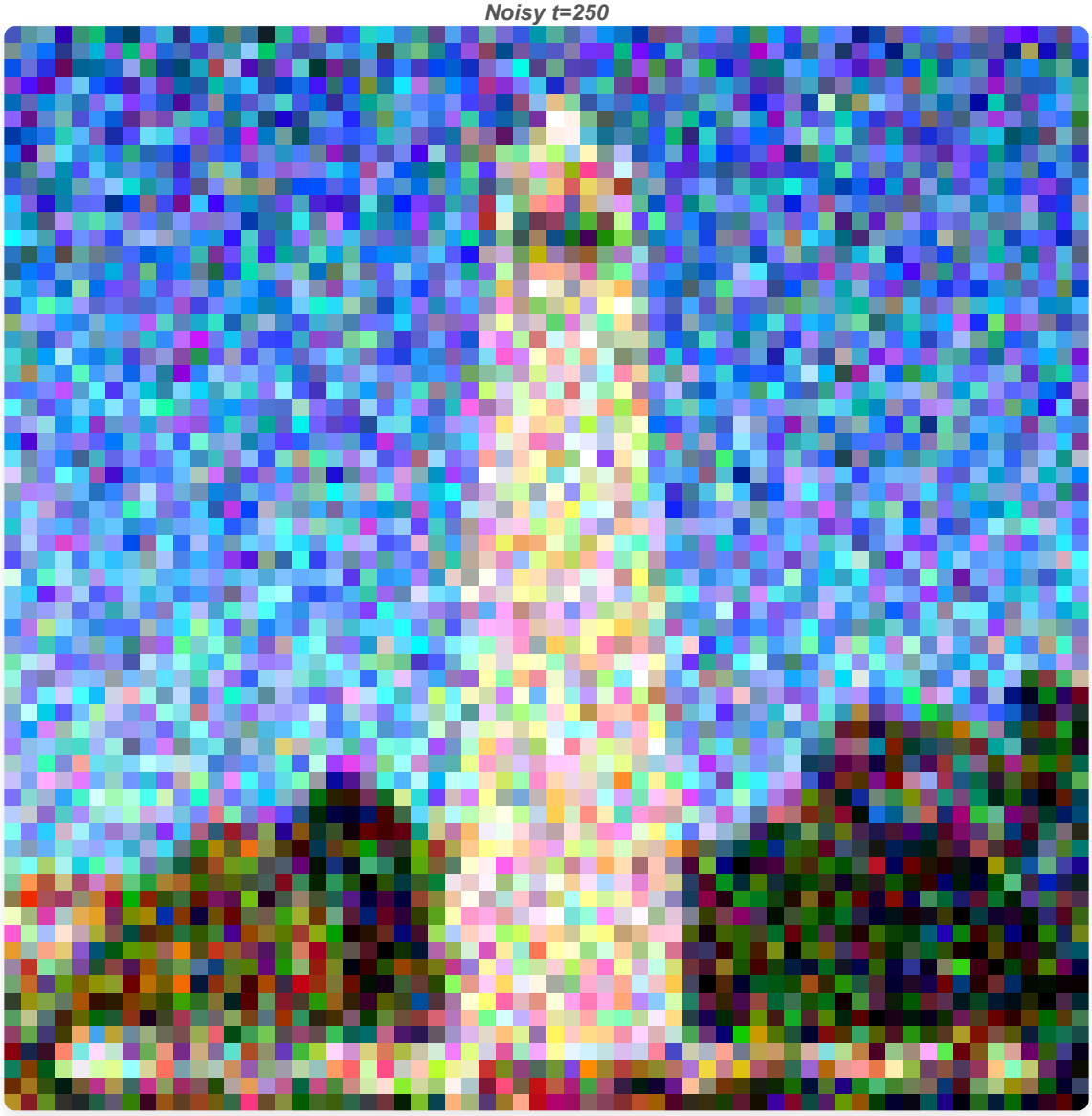
Berkeley Campanile (original)	*t=250*	*t=500*	*t=750*



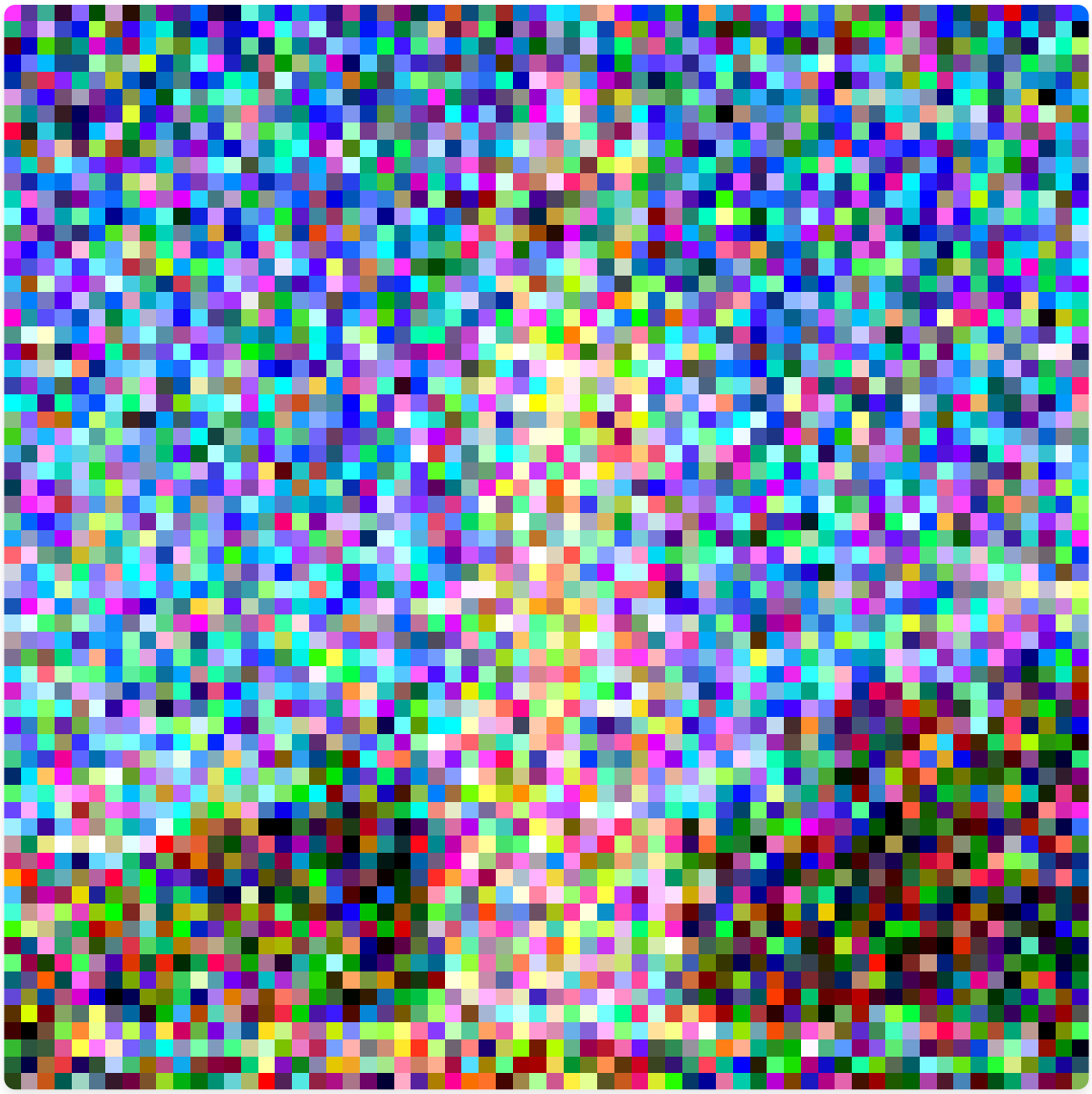
1.2 Classical Denoising

We can attempt to denoise the image by convolving it with the Gaussian kernel, effectively blurring it. This works somewhat, but does not look very nice. I use a kernel size of 7 and sigma of 2 for the Gaussian kernel.

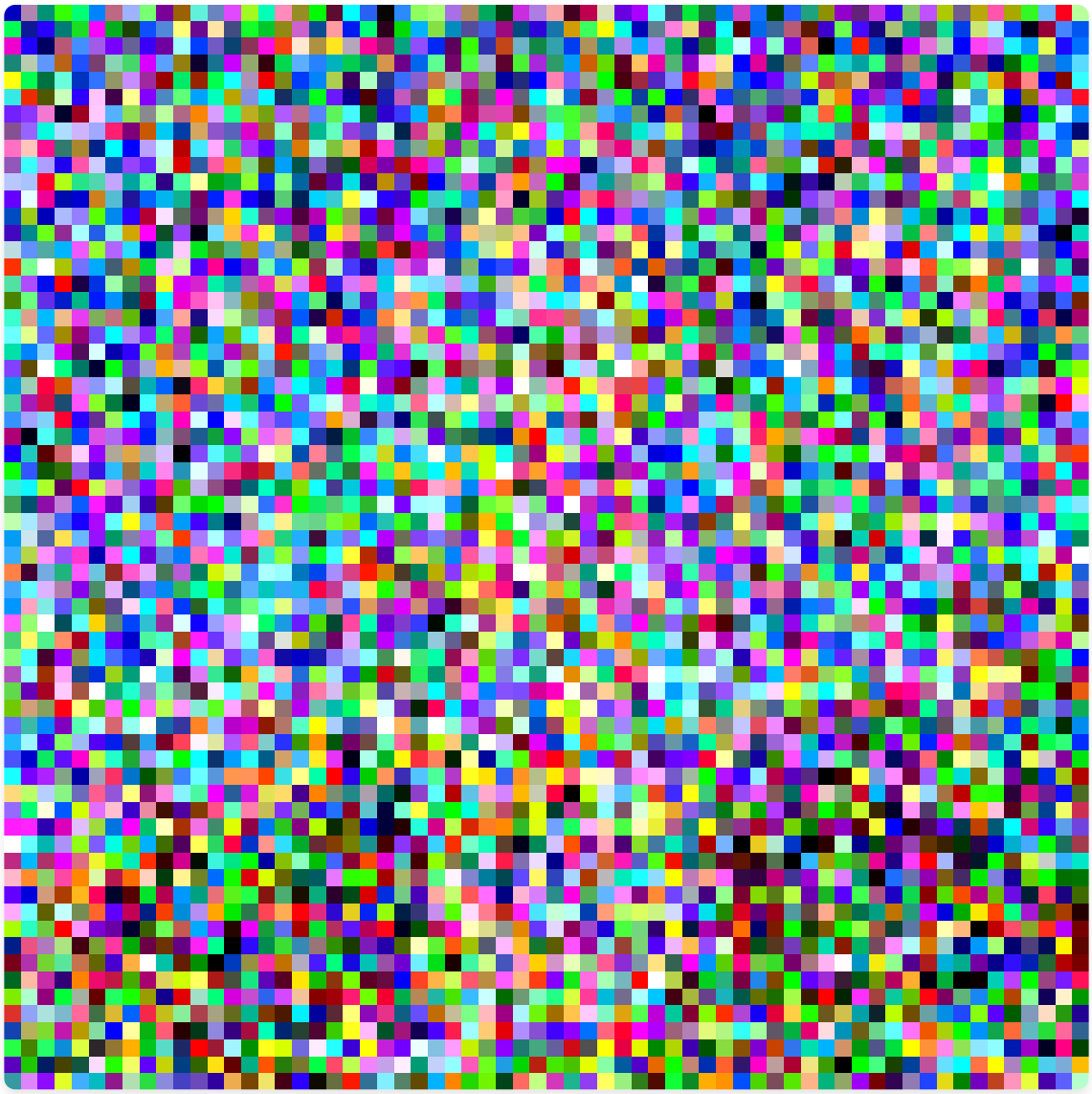
Noisy t=250



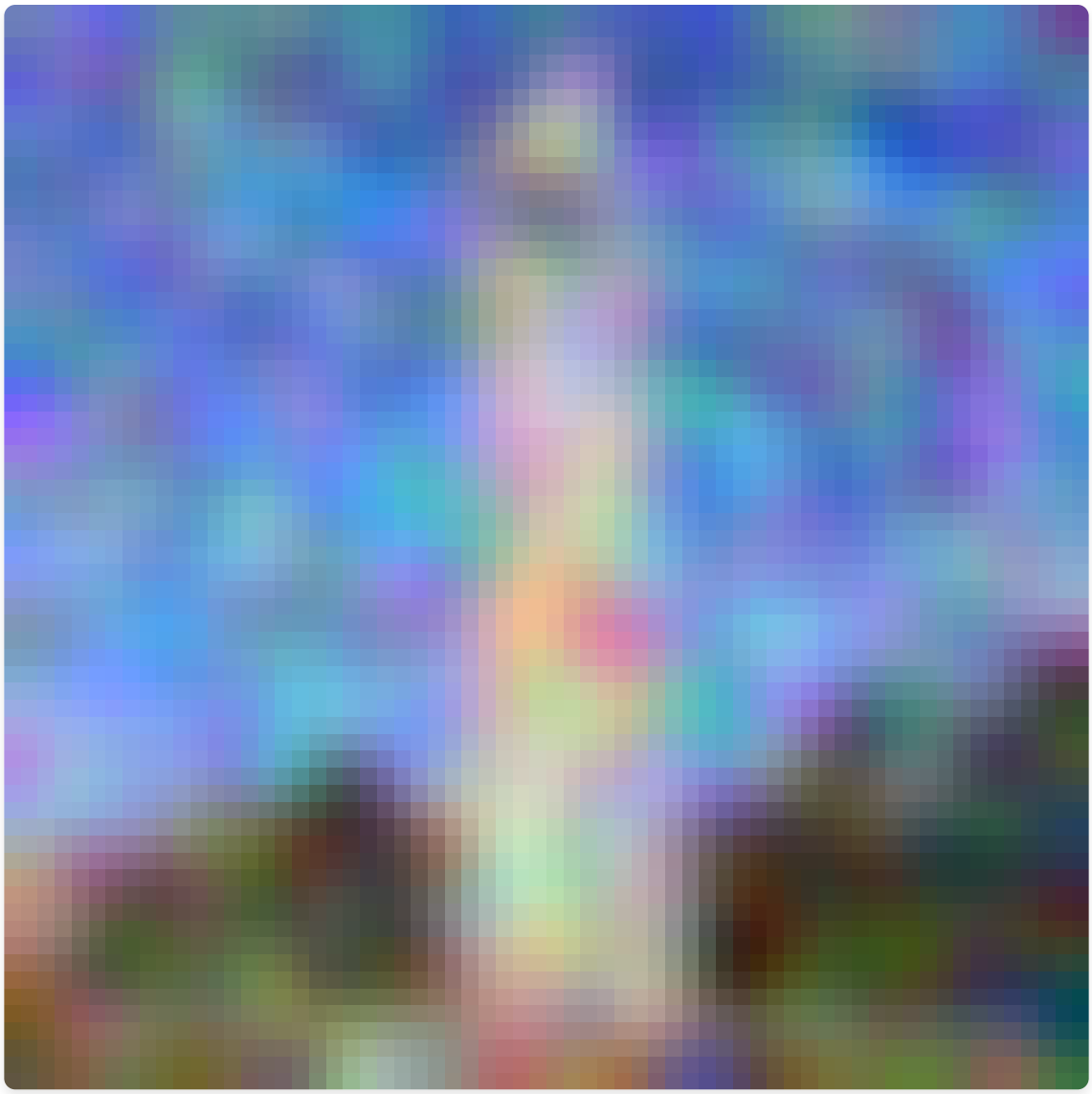
Noisy t=500



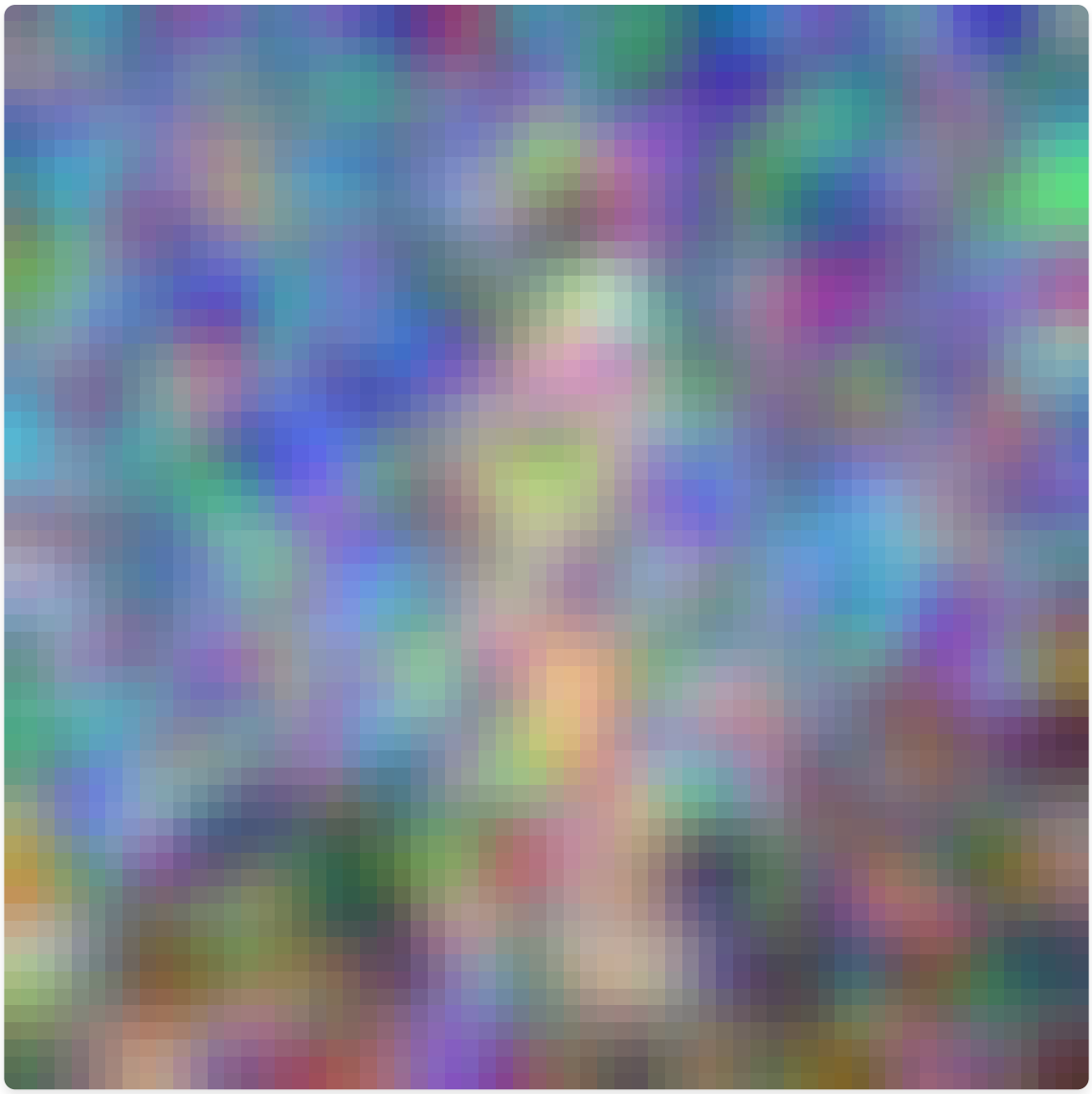
Noisy t=750



Blurred t=250



Blurred t=500



Blurred t=750

1.3 Implementing One-Step Denoising

We can use the pretrained DeepFloyd model's stage 1 denoiser to predict and remove the noise added to the image in a single step. We can solve the equation in the previous section for x0 and use this to scale and subtract the estimated noise:

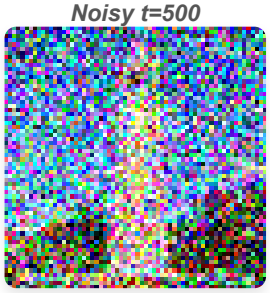
$$\mathrm{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon}{\sqrt{\bar{\alpha}_t}}$$

The resulting images are displayed below for t=250, 500, and 750:

Berkeley Campanile (original)

Noisy t=250

Noisy t=500



Noisy t=750

Original (unchanged) *Denoised t=250* *Denoised t=500* *Denoised t=750*

We can see that the denoising UNet does a lot better of a job of projecting the image onto the manifold of natural images, although more of the details are hallucinated the more noisy the original image is.

1.4 Implementing Iterative Denoising

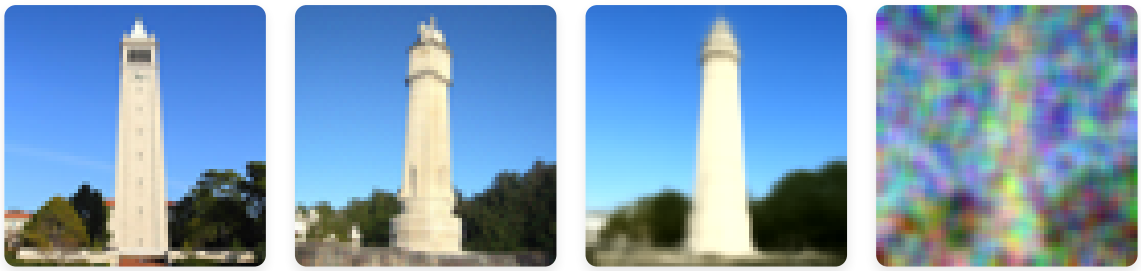
Diffusion models work better when denoised iteratively. We can skip some steps to speed up inference, stepping from T=990 to T=0 in steps of 30. The update rule taking the stride into account is below:

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

t' is the next timestep (less than the current timestep t), alpha t = alpha bar t / alpha bar t', beta = 1 - alpha t, and x0 is the current estimate of the clean image using the equation for x0 in section 1.3. v sigma is the random noise DeepFloyd predicted.

The results are displayed below with noise from i_start = 10 and timestep[10] and compared to the previous 2 methods we tried above.

Noisy Campanile at t=690 *Noisy Campanile at t=540* *Noisy Campanile at t=390* *Noisy Campanile at t=240* *Noisy Campanile at t=90*



Original *Iterative Denoising* *One-Step Denoising* *Gaussian Blurred*

We can see that the iterative generation produces a higher quality image with more details, however most of these details are hallucinated.

1.5 Diffusion Model Sampling

We can also repeat the iterative process in 1.4 and denoise images from randomly sampled noise and i_start=0, effectively projecting the noisy images onto the manifold of images learned by the model. We use a "null" prompt that doesn't have any specific meaning, "a high quality photo", in the following generations from random noise (upscaled):

Sample 1 *Sample 2* *Sample 3* *Sample 4* *Sample 5*

1.6 Classifier-Free Guidance (CFG)

The generated images are somewhat nonsensical. To improve image quality at the expense of image diversity, we implement Classifier-Free Guidance. We compute a conditional and unconditional noise estimate epsilon c and epsilon u, and set our new noise estimate to be:

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

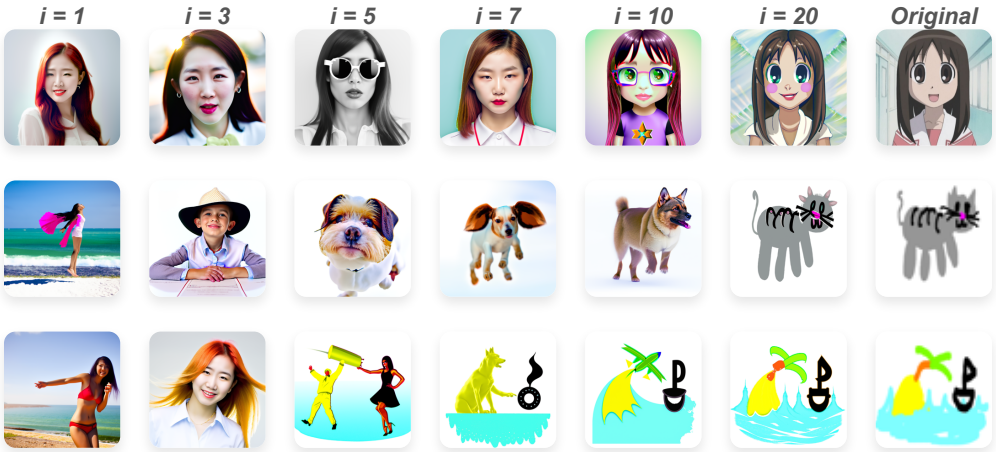
Where gamma is a scaling factor. We set gamma = 7 for the following generations, and use a nuill prompt of "" for the unconditional guidance noise estimate. The upscaled results are displayed below:

1.7 Image-to-image Translation

We can make edits to an existing image by taking an image, noising it, and forcing it back to the image manifold without conditioning, following the SDEdit algorithm. We run forward with starting indices of [1,3,5,7,10,20] and apply it to the Campanile image, as well as 2 pictures I've taken (one of Osaka castle and another of me and my friends), displaying the upsampled results below (original images are shown at 256x256 resolution for comparison but they are inputted to the model as 64x64):

1.7.1 Editing Hand-Drawn and Web Images

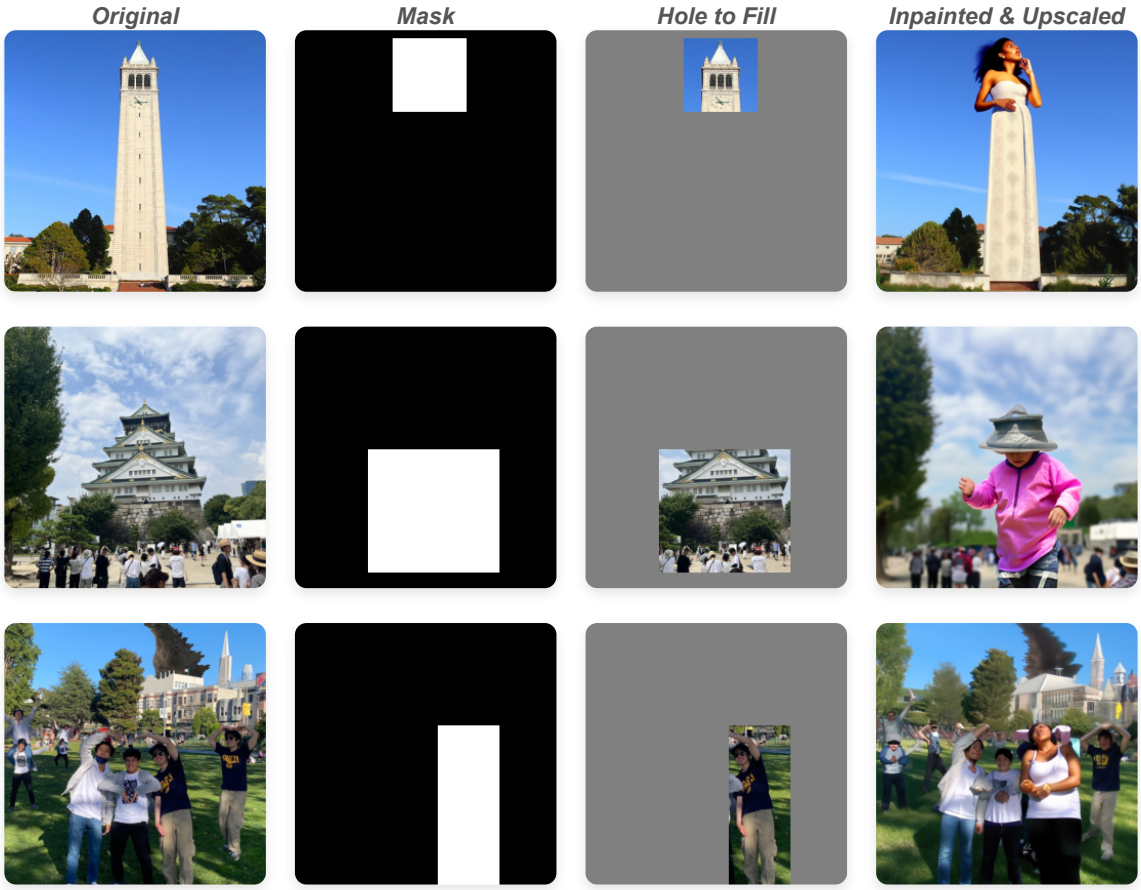
We can use the above procedure to project nonrealistic images onto the natural image manifold. Displayed below are 1 result from the Internet (Osaka from Azumanga Daioh) as well as 2 of my own sketches.



1.7.2 Inpainting

We can direct where the model should hallucinate new information by using a binary mask with the original image. Given an image x_orig and a binary mask m, we create a new image with the same content where m is 0 and new content where m is 1. We use same diffusion denoising loop except updating xt with the following after every step:

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



Notably, upscaling ends up modifying features from the original image. In general, I believe that this is helpful in making a more cohesive image, however in some cases we do not want this to happen. In Part 2: Bells and Whistles I examine using a binary mask and our old friend Laplacian Blending with the 256x256 image and the upscaled image to keep details from the original image.

1.7.3 Text-Conditional Image-to-image Translation

We repeat the process in 1.7 and 1.7.1, perturbing the images with noise, except this time we guide it with a text prompt "a rocket ship" instead of "a high quality photo" to guide generation towards the image subspace of rockets.

1.8 Visual Anagrams

We now proceed to make optical illusions using our diffusion models. In this section, we create an image that looks like one prompt but when flipped upside down looks like another prompt. In order to do this, we compute 2 separate noise estimates using the first prompt and the flipped image with the second prompt and average them t:

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

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

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

When we upscale the images, we use the "null" prompt "a high quality photo".

an oil painting of people around a campfire

a photo of a hipster barista

an oil painting of a snowy mountain village

an oil painting of an old man

a photo of a dog

a lithograph of waterfalls

1.10 Hybrid Images

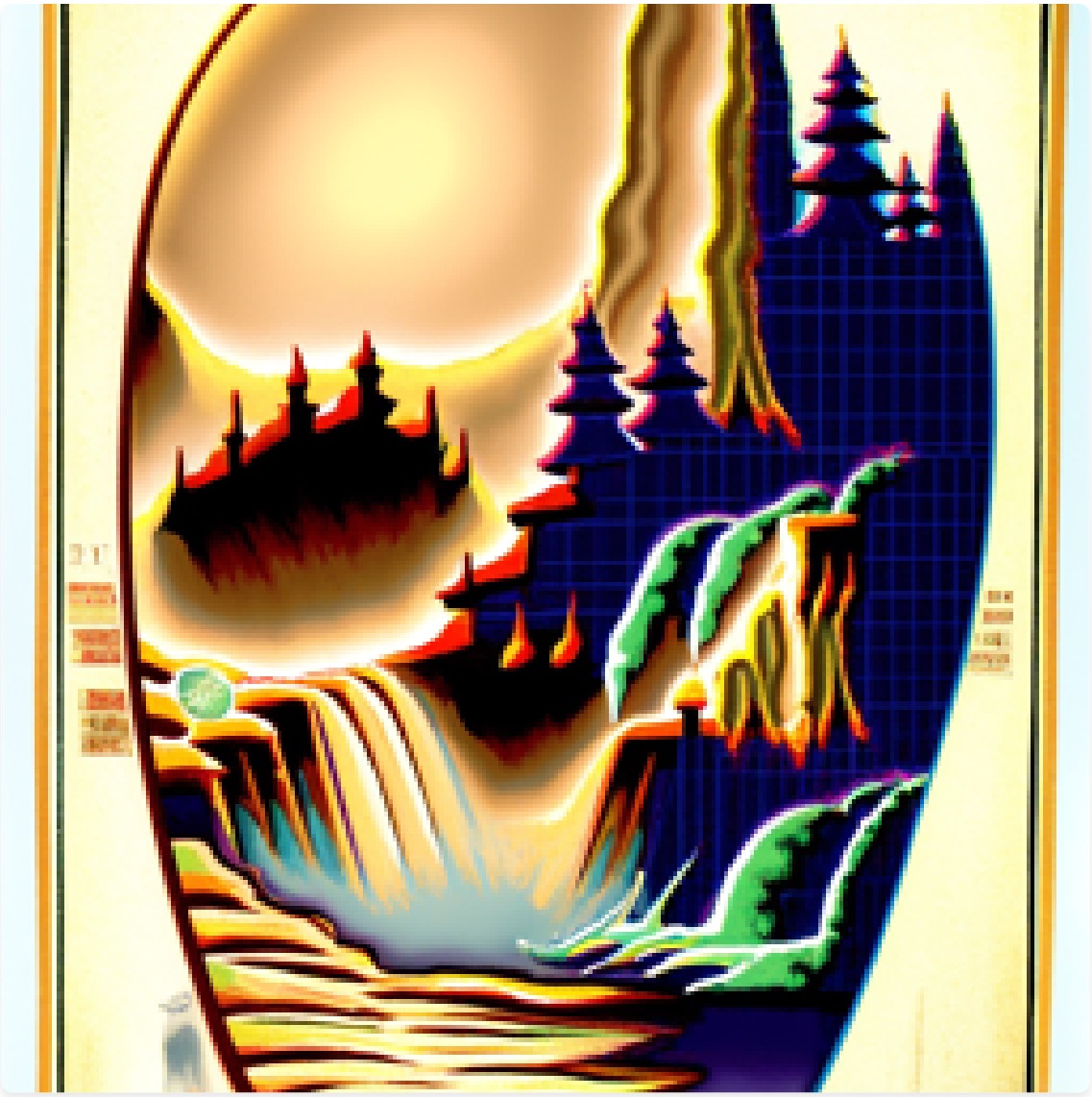
We can create hybrid images in a similar process to 1.8, by combining noise estimates. This time, we perform a lowpass filter (convolve with gaussian) for one prompt's noise estimate, and a high pass (laplacian filter) of the second prompt's noise estimate:

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

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

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

Hybrid skull and waterfall



Hybrid dog and Amalfi Coast

Hybrid pencil and man in a hat

Part 2: Bells and Whistles

Recall in part 1.7.2, the upscaling of inpainted images results in details the original image also being hallucinated. I experiment with using multi-resolution blending to combine the upscaled in-painted image with the 256x256 original image to preserve the original image's details as opposed to simply masking the upscaled in-painted image over the original image.

Using multi-resolution blending, we find marginal improvements over naive cropping, although the efficacy definitely depends on the images used.

Part B: Diffusion Models from Scratch!

Overview

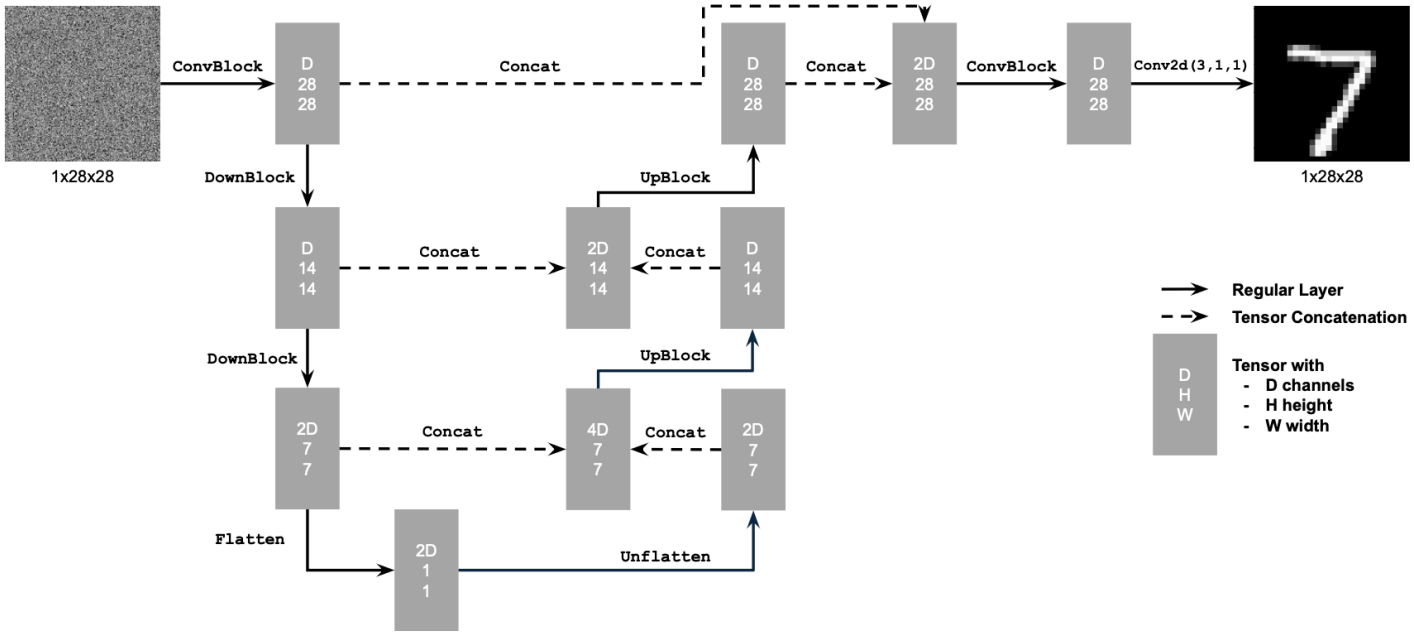
In the second part of this project we implement, train, and test the UNet architecture on denoising and generating images using the MNIST dataset. Time Conditioning and Class Conditioning with classifier-free guidance are incorporated into our model.

Part 1: Training a Single-Step Denoising UNet

We first optimize a simple one-step denoiser over the L2 loss between the model's prediction and the clean image.

1.1 Implementing the UNet

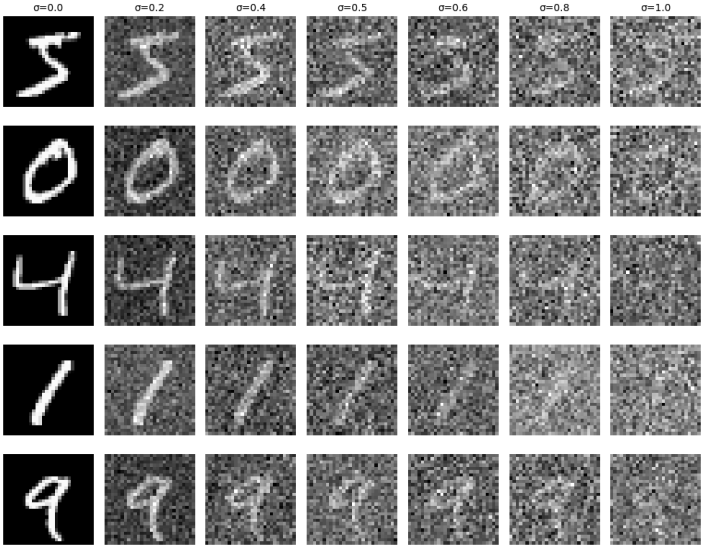
We implement the following architecture for an Unconditional UNet:



1.2 Using the UNet to Train a Denoiser

For each training batch, we generate z=x + sigma * epsilon, where epsilon is sampled from N(0, I). We input z into our model, and train it using the L2 distance between the model's prediction for z and the original image x.

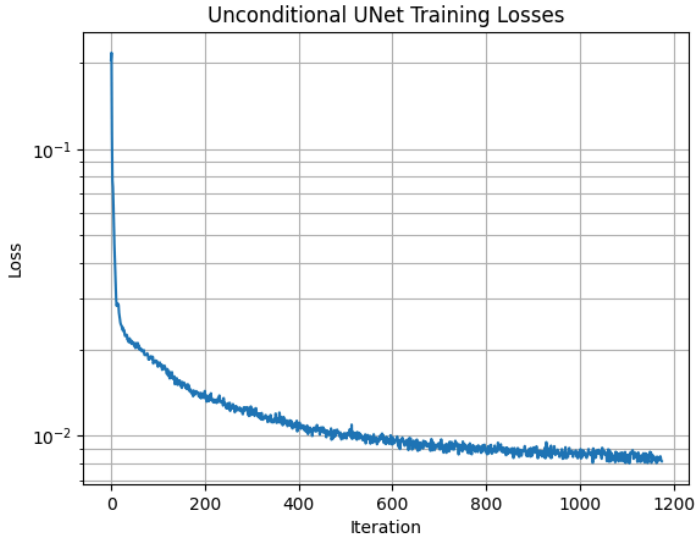
We visualize the varying levels of noise on the MNIST digits below:



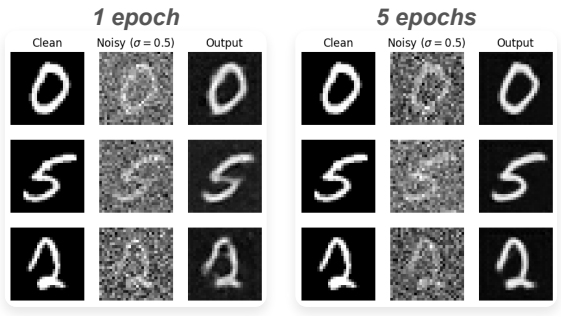
1.2.1 Training

We train the model to denoise noisy images with sigma=0.5, with batch size 256, 5 epochs, hidden dimension 128, and adam optimizer with learning rate 1e-4.

The loss curve is plotted below:



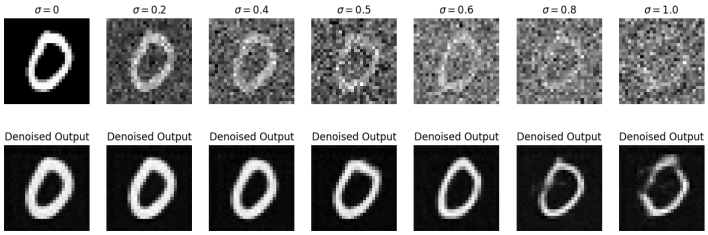
We visualize the denoised results on a small test sample batch for 1 epoch and 5 epochs:



This UNet does a pretty good job at denoising the images at the noise level it is trained on.

1.2.2 Out-of-Distribution Testing

In this section, we see how well the denoiser does at different noise levels, for sigma in [0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0].



We can observe that the model generally does worse for noise levels it was not trained on, but this effect is most pronounced at higher noise levels as opposed to lower noise levels.

Part 2: Training a Diffusion Model

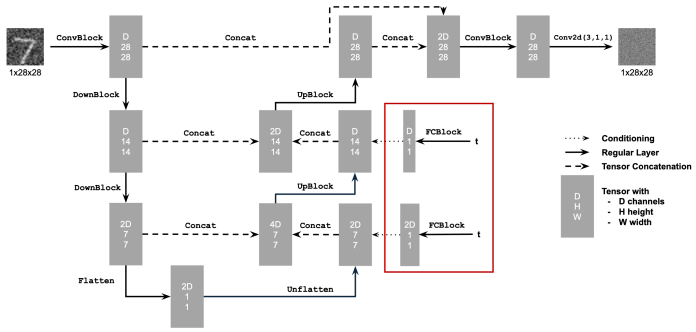
Instead of training with the L2 loss between our model's predicted image and the clean image, we instead iteratively denoise the image by training the model instead on the L2 loss between the model's prediction and the noise added to the image.

We use the same formula as the forward process in part A to generate noisy images. Our betas are set to be evenly spaced between beta0=0.0001 and betaT=0.02 for beta between 1 and T-1, and the alphas and alpha bars are defined based on the betas as in part A.

We use T=300 instead of T=1000 in part A as MNIST digits are relatively simple.

2.1 Adding Time Conditioning to UNet

We condition our UNet model with the time t that the model should be trained on by inserting FC blocks in the following manner, normalizing t before embedding it as t/T:



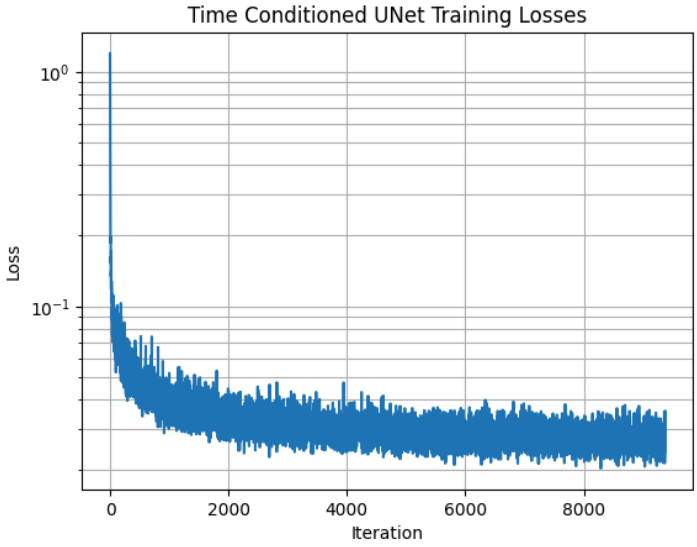
2.2 Training the UNet

We follow the following algorithm to train our time-conditioned UNet, sampling a random image from the training set, a random t, and training the denoiser to predict the noise in xt:

Algorithm 1 Training

1: Precompute $\bar{\alpha}$
2: **repeat**
3: $\mathbf{x}_0 \sim$ clean image from training set
4: $t \sim \text{Uniform}(\{1, \ldots, T\})$
5: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
6: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$
7: $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$
8: Take gradient descent step on $\nabla_\theta \left\|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}\right\|^2$
9: **until** happy

The loss curve for this model is displayed below:



2.3 Sampling from the UNet

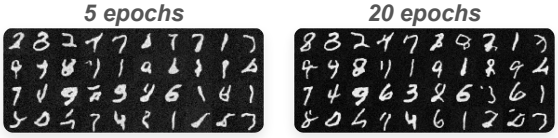
We follow a similar sampling process to part A, except we don't predict the variance:

Algorithm 2 Sampling

Precompute β, α, and $\bar{\alpha}$
$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for t from T to 1, step size -1 **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(\mathbf{x}_t, t))$ ▷ See part A of project
  $\mathbf{x}_{t-1} = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \sqrt{\beta_t}\,\mathbf{z}$
end for
return $\mathbf{x}_0$

Our sampling results for the time-conditioned UNet for 5 and 20 epochs are displayed below:

5 epochs *20 epochs*



We can see that the model does a bit poorly, generating some recognizable digits from random noise, however there are also a lot of random strokes and figures that do not look like digits.

2.4 Adding Class-Conditioning to UNet

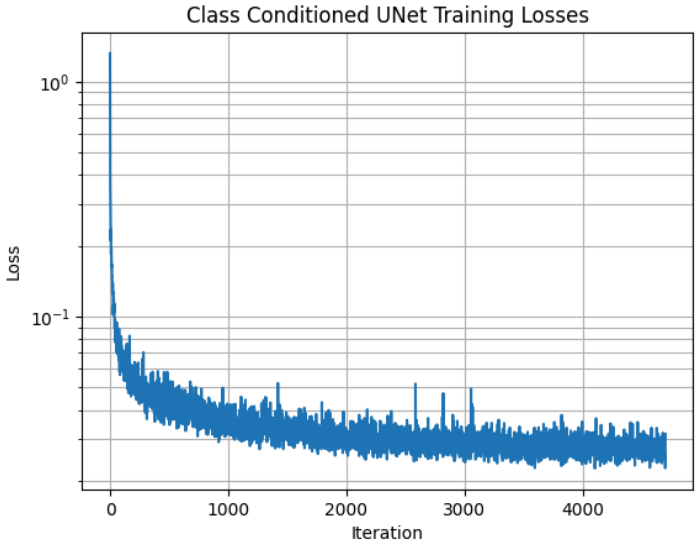
We can get better results by also conditioning UNet on the class of the digit 0-9. This requires us to embed 2 more FCBlocks to the UNet (in the same locations that the blocks for t are placed, except multiplying the model outputs instead ofbeng added onto them).

To make the model still work without being conditioned on the class, we also implement dropout, masking the class conditioning vector c to 0 for 10% of the training time. The class conditioned training algorithm is displayed below:

Algorithm 3 Class-Conditioned Training

Precompute $\bar{\alpha}$
repeat
  $\mathbf{x}_0,\ c \sim$ clean image and label from training set
  Make c into a one-hot vector
    (with probability p_{uncond} set c to zero-vector).
  $t \sim \text{Uniform}(\{1, \ldots, T\})$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$
  $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t, c)$
  Take gradient descent step on
    $\nabla_\theta \|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}\|^2$
until happy

Our training loss curve for this model is as follows:



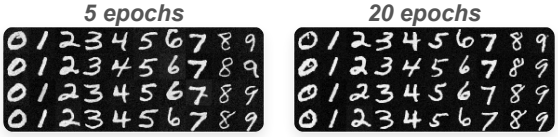
2.5 Sampling from the Class-Conditioned UNet

The sampling process is the same as part A, and we use classifier-free guidance with gamma=5.0 to improve the conditional results. The algorithm is displayed below:

Algorithm 4 Class-Conditioned Sampling

input: one-hot vector c, classifier-free guidance scale γ
$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for t from T to 1, step size -1 **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\epsilon_u = \epsilon_\theta(\mathbf{x}_t, t, 0)$
  $\epsilon_c = \epsilon_\theta(\mathbf{x}_t, t, c)$
  $\epsilon = \epsilon_u + \gamma(\epsilon_c - \epsilon_u)$ ▷ Classifier-free guidance
  $\hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon)$
  $\mathbf{x}_{t-1} = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \sqrt{\beta_t}\,\mathbf{z}$
end for
return $\mathbf{x}_0$

We visualize our results for all 10 digits below for epochs 5 and 20:



We can see that the results for both epochs are relatively good, but the 20 epoch model tends to keep more finer details.