CS 180: Project 5

Fun With Diffusion Models!

Meenakshi Mittal

Project 5a: The Power of Diffusion Models!

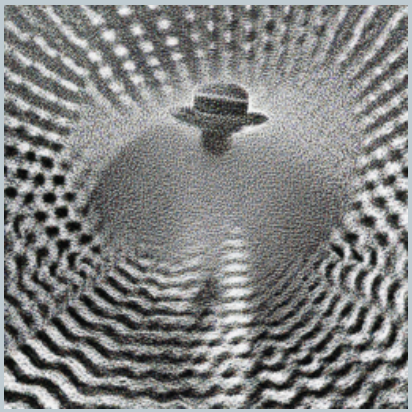
In this part of the project, we will play around with a pre-trained diffusion model, DeepFloyd IF, accessed via Hugging Face. We use random seed 180 (but some images displayed here were generated after rerunning the model a few times).

Part 0: Setup

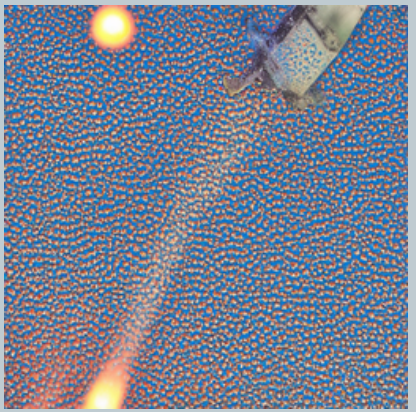
First, we see what this model can do. We have it generate 64x64 images on 3 prompts, and then pass these images through again to upscale them to 256x256. We test 3 different numbers of inference steps:

5 inference steps:

"an oil painting of a snowy mountain village"



"a man wearing a hat"



"a rocket ship"

With only 5 inference steps, the generated images have quite poor quality. They all share an odd "pointillism" effect, with lots of tiny points obscuring the images. The mountain village adheres to the prompt the most, although it doesn't really look like an oil painting. The pointillism almost looks like snow. The man wearing a hat is honestly very artistic and impressionistic, and I like it a lot even though it isn't very accurate. The rocket ship appears to be a mostly blank image with the rocket mostly off the screen and its smoke trailing behind it.

20 inference steps:

"an oil painting of a snowy mountain village"

"a man wearing a hat"

"a rocket ship"

20 inference steps is looking a lot better. The mountain village is very easy to make out now, with bright colors and a playful art style. It looks more like an oil painting. The man with a hat is less artsy and much more realistic now. There is a bit of odd contrasting and blurriness that give an uncanny valley feeling. The rocket ship is much more clear to see in this image, again with a rather childish art style.

50 inference steps:

"an oil painting of a snowy mountain village"

"a man wearing a hat"

"a rocket ship"

At 50 inference steps, our images look really good. The mountain village is artsy and quite beautiful. The man wearing a hat is a lot sharper and even more realistic than the previous one. The rocket itself is of somewhat similar quality, but there is a very realistic rocky surface below it now.

Part 1: Sampling Loops

Another thing we can experiment with is how good this model is at denoising images. To do this, we first need to add noise to some image.

1.1: Implementing the Forward Process

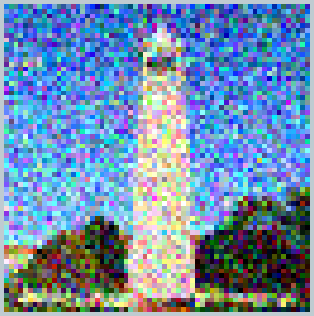
We ideally want different "levels" of noise, so we will iteratively add noise to the image in our forward process using the following formula:

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

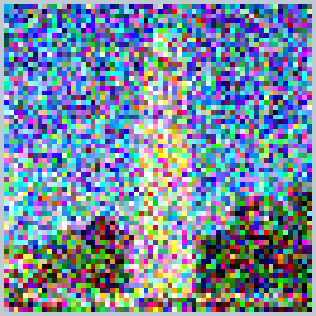
Here, alpha_t_bar was chosen by the people who trained the model, and t ranges from 0 to 999. At t=0, the

image is clean, and large t corresponds to more noise. We will use an image of the UC Berkeley bell tower, the Campanile. Here is the original image, along with the noisy versions of it at t in [250, 500, 750]:

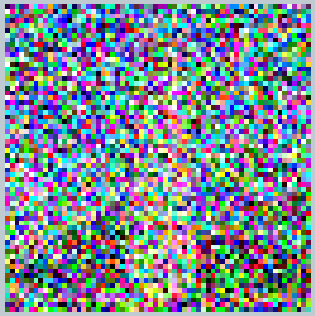
Berkeley Campanile



Noisy Campanile at t=250



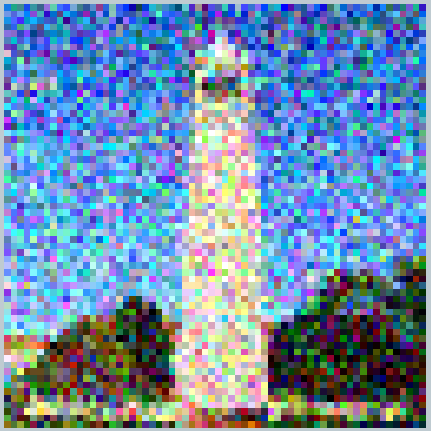
Noisy Campanile at t=500



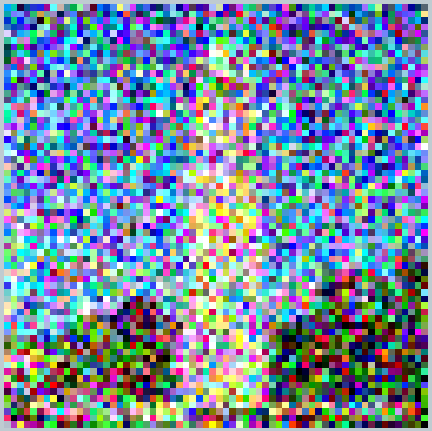
Noisy Campanile at t=750

1.2 Classical Denoising

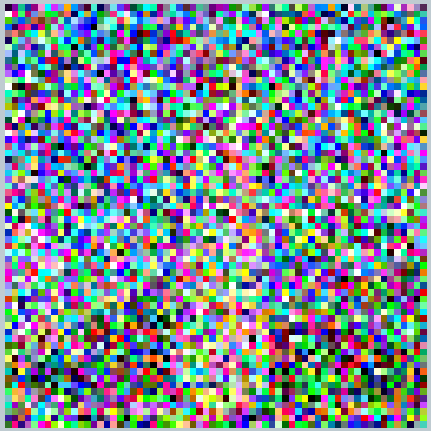
Let's try to denoise these images using classical methods. We will try our best to use simple Gaussian blur filtering to remove the noise. For timesteps 250, 500, and 750, I used Gaussian kernel sizes of 7, 11, and 15, respectively. The results leave much to be desired:



Noisy Campanile at t=250

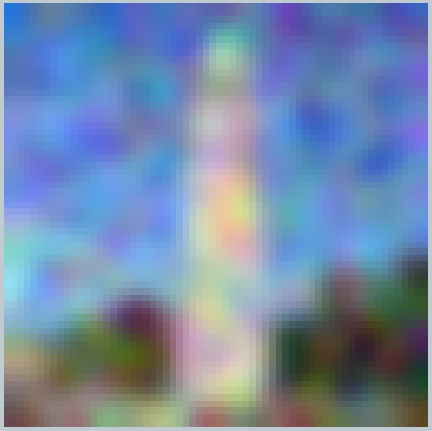


Noisy Campanile at t=500

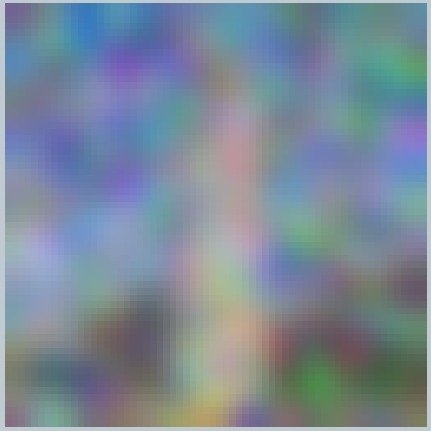


Noisy Campanile at t=750

Gaussian Blur Denoising at t=250



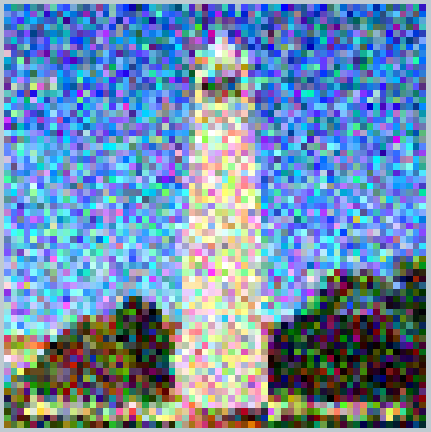
Gaussian Blur Denoising at t=500



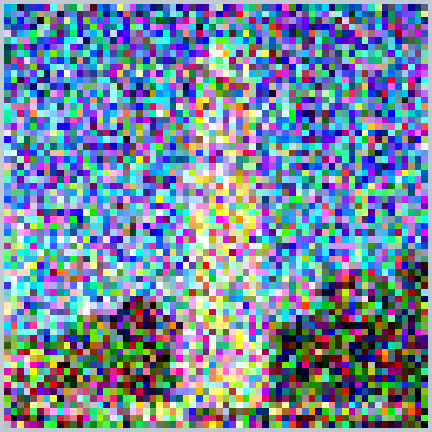
Gaussian Blur Denoising at t=750

1.3: One-Step Denoising

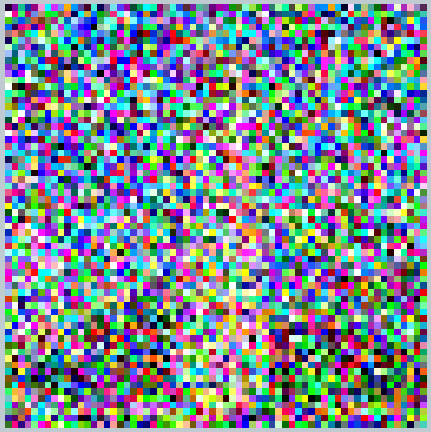
Now we will try using a pre-trained diffusion model to denoise these images. The model we are using has been trained on a massive dataset of noisy images, and is optimized for predicting the noise in an image. We can simply pass our images through this model to get their noise estimates, and then use our forward equation from above to retrieve an estimate of the cleaned image (x0). We rewrite the equation in terms of x0, the clean image, and substitute our noise estimate for epsilon. The results from this method are a significant step up from the Gaussian blurring approach. However, while the results are higher quality, it is worth noting that the model appears to be "making up" some parts of the images that were previously obscured by the noise. For instance, the denoised image from t=750 looks quite different from the actual Campanile.



Noisy Campanile at t=250



Noisy Campanile at t=500



Noisy Campanile at t=750

One-Step Denoising at t=250

One-Step Denoising at t=500

One-Step Denoising at t=750

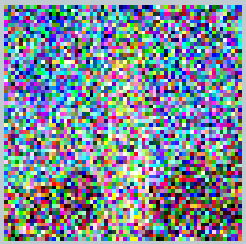
1.4 Iterative Denoising

The denoising UNet seems to do a petty decent job. However, we can try to make it produce even better results. Instead of denoising in a single step, we will try to iteratively denoise the image. We will start with a noisy image at t=690, and iteratively predict the image at 30 timesteps prior. We will use the following formula to accomplish this:

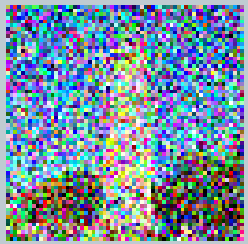
$$x_{t'} = \frac{\sqrt{\bar{\alpha}_{t'}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t'})}{1-\bar{\alpha}_t}x_t + v_\sigma$$

- t is our "starting" timestep.
- t' is the less noisy timestep that we are predicting.
- alpha_t_bar is defined as it was previously.
- alpha_t = alpha_t_bar / alpha_t'_bar.
- beta_t = 1-alpha_t.
- x0 is an estimate of the clean image, using the one-step denoising above.

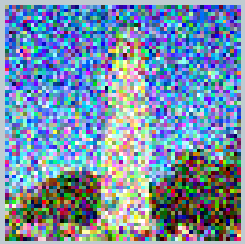
Below, we see 5 steps of the iterative denoising process. We also see each of our 3 denoising methods compared side by side. For the Gaussian blurred one, I chose a kernel size of 13.



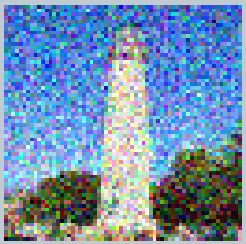
Noisy Campanile at t=690



Noisy Campanile at t=540



Noisy Campanile at t=390



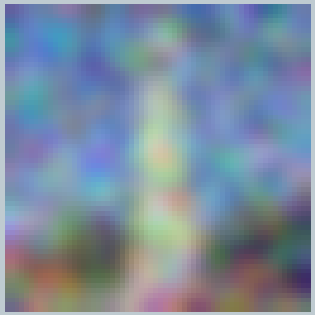
Noisy Campanile at t=240

Noisy Campanile at t=90

Berkeley Campanile

Iteratively Denoised Campanile

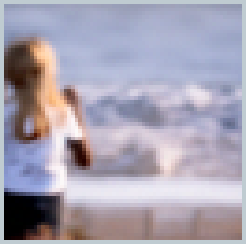
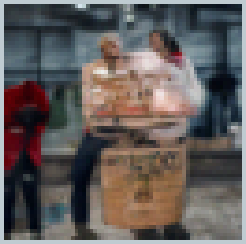
One-Step Denoised Campanile

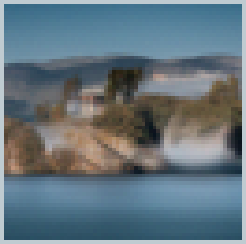
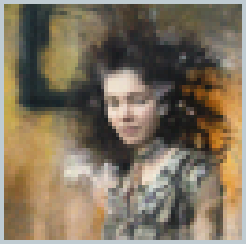


Gaussian Blurred Campanile

1.5 Diffusion Model Sampling

Now we will move on from the Campanile and see if we can get our model to generate "high quality" images from pure noise. We will use a similar iterative denoising process, except now starting from t=990 and using the prompt "a high quality photo":

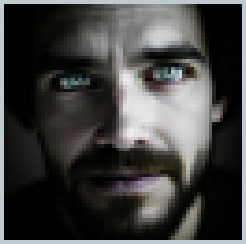
Sample 1 Sample 2 Sample 3 Sample 4 Sample 5

1.6 Classifier-Free Guidance (CFG)

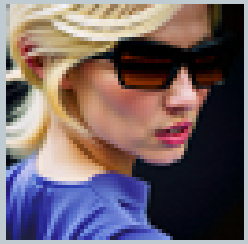
The images above are decent, but a bit strange and hard to make out. To improve the quality of our images, we will use a technique called "classifier-free guidance", or CFG. Here, we compute both a conditional and unconditional noise estimate, and then combine them using the following equation:

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

For $\gamma = 0$ we get an unconditional noise estimate and for $\gamma = 1$ we get the conditional noise estimate. For some unknown reason, setting $\gamma > 1$ gives much higher quality images. The images appear to have higher contrast, more vivid colors, and just better composition overall. Here is a sample of images with $\gamma = 7$:







Sample 1 Sample 2 Sample 3 Sample 4 Sample 5

1.7 Image-to-image Translation

One way to think about the denoising process from earlier is that it "forces" a noisy image back onto the manifold of natural images. We can have some fun by denoising images using iterative CFG denoising, allowing the model to creatively fill in the unknown parts itself. Below are 3 images that I applied this approach, known as the SDEdit algorithm, at varying noise levels:

Campanile:







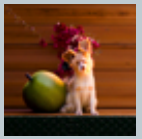
Original Image SDEdit, i_start=1 SDEdit, i_start=3 SDEdit, i_start=5 SDEdit, i_start=7 SDEdit, i_start=10 SDEdit, i_start=20

Vase with fruit:









Original Image	SDEdit, i_start=1	SDEdit, i_start=3	SDEdit, i_start=5	SDEdit, i_start=7	SDEdit, i_start=10	SDEdit, i_start=20

Astronaut:







Original Image	SDEdit, i_start=1	SDEdit, i_start=3	SDEdit, i_start=5	SDEdit, i_start=7	SDEdit, i_start=10	SDEdit, i_start=20

1.7.1 Editing Hand-Drawn and Web Images

We will repeat the process above with less realistic images, i.e., animated images from the web and our own hand-drawn images:

Simba from The Lion King:







Original Image | SDEdit, i_start=1 | SDEdit, i_start=3 | SDEdit, i_start=5 | SDEdit, i_start=7 | SDEdit, i_start=10 | SDEdit, i_start=20

Hand-drawn image of a beach:



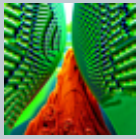




Original Image | SDEdit, i_start=1 | SDEdit, i_start=3 | SDEdit, i_start=5 | SDEdit, i_start=7 | SDEdit, i_start=10 | SDEdit, i_start=20

Hand-drawn image of a forest:




Original Image	SDEdit, i_start=1	SDEdit, i_start=3	SDEdit, i_start=5	SDEdit, i_start=7	SDEdit, i_start=10	SDEdit, i_start=20

1.7.2 Inpainting

We can also use these ideas to "inpaint" an image. In other words, we can keep some part of an image the same, and let the diffusion model replace the other part. We do this by first creating a mask, and then running the regular CFG denoising loop with the following formula applied at each step:

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

This formula leaves everything inside the edit mask alone, but replaces everything outside the mask with our original image with the correct noise level. Here are some inpainting examples:

Campanile:


Original Image


Mask

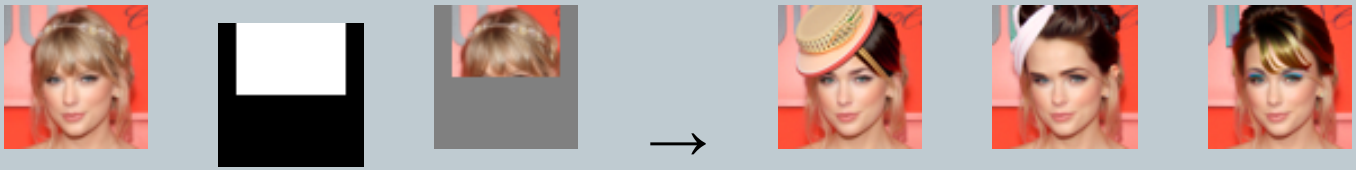

Hole to Fill


Inpainted 1


Inpainted 2

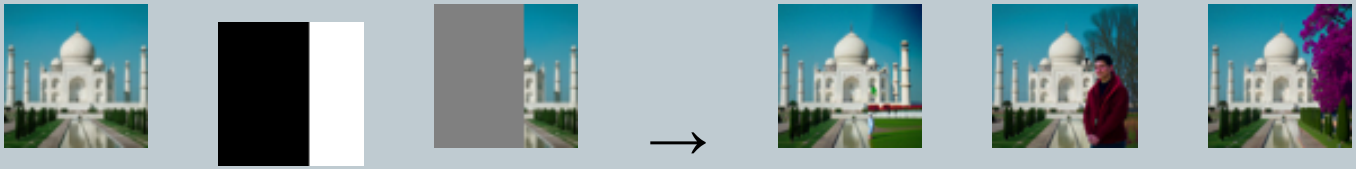

Inpainted 3

Taylor Swift:



Original Image | Mask | Hole to Fill | Inpainted 1 | Inpainted 2 | Inpainted 3

Taj Mahal:



Original Image | Mask | Hole to Fill | Inpainted 1 | Inpainted 2 | Inpainted 3

1.7.3 Text-Conditional Image-to-image Translation

Until now, we have always been using the prompt "a high quality photo". However, we can influence the projection of the image onto the natural image manifold by providing it with a natural language prompt to adhere to. Here are some examples of this:

Campanile, with prompt "a rocket ship":



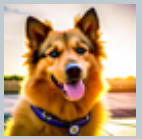
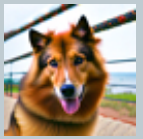




Original Image | SDEdit, i_start=1 | SDEdit, i_start=3 | SDEdit, i_start=5 | SDEdit, i_start=7 | SDEdit, i_start=10 | SDEdit, i_start=20

Cat, with prompt "a photo of a dog":



Original Image | SDEdit, i_start=1 | SDEdit, i_start=3 | SDEdit, i_start=5 | SDEdit, i_start=7 | SDEdit, i_start=10 | SDEdit, i_start=20

Billie Eilish, with prompt "a man wearing a hat":







Original Image	SDEdit, i_start=1	SDEdit, i_start=3	SDEdit, i_start=5	SDEdit, i_start=7	SDEdit, i_start=10	SDEdit, i_start=20

1.8 Visual Anagrams

Now we can try some really cool techniques. We will create visual anagrams that change appearance when flipping upside down. We can do this using our current prompted denoising approach. At every step, we find the noise estimate of the image using one prompt, and we also flip it upside down and find the noise estimate using a different prompt. Then we average these 2 noise estimates and take a denoising step. Here is the algorithm used to compute the noise estimate:

$$\epsilon_1 = \text{UNet}(x_t, t, p_1)$$
$$\epsilon_2 = \text{flip}(\text{UNet}(\text{flip}(x_t), t, p_2))$$
$$\epsilon = (\epsilon_1 + \epsilon_2)/2$$

I had to try many different combinations of prompts and rerun the model several times to get good results. Here are 3 best visual anagrams I could create using this algorithm:

"an oil painting of people around a campfire"

→

"an oil painting of an old man"

"an oil painting of a young woman"

→

"an oil painting of an old lady"

"an oil painting of a ballet dancer with a tutu"

→

"an oil painting of a ship in the ocean"

1.9 Hybrid Images

We can use a somewhat similar process to create "hybrid images", similar to the ones we created in Project 2. We will use a method called Factorized Diffusion, where we compute noise estimates of the image using 2 different prompts, and then add the lowpass version of one with the highpass version of the other. Here is the algorithm we use:

$$\epsilon_1 = \text{UNet}(x_t, t, p_1)$$
$$\epsilon_2 = \text{UNet}(x_t, t, p_2)$$
$$\epsilon = f_{\text{lowpass}}(\epsilon_1) + f_{\text{highpass}}(\epsilon_2)$$

I used a Gaussian blur with kernel size 33 and sigma 2 for the lowpass filter. For the highpass filter, I use the same Gaussian filter and subtract the resulting image from the original. I again had to try many different combinations of prompts and rerun the model several times to get good results. Here are the 6 best hybrid images I could create, with 2 for each pair of prompts:

Highpass prompt: "a lithograph of waterfalls"

Lowpass prompt: "a lithograph of a skull"

Waterfalls

Skull

Waterfalls

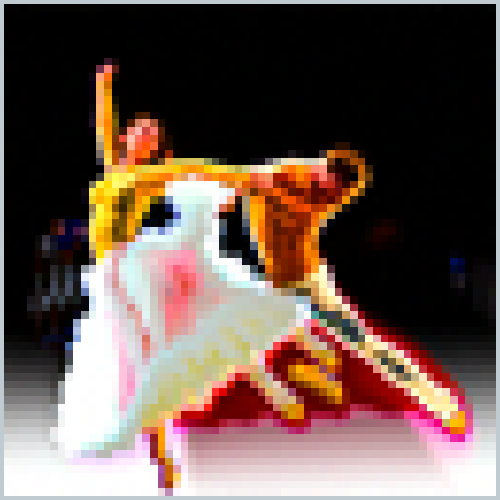
Skull

Highpass prompt: "an oil painting of people dancing"

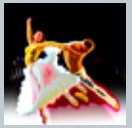
Lowpass prompt: "a photo of a cat"

People Dancing

Cat



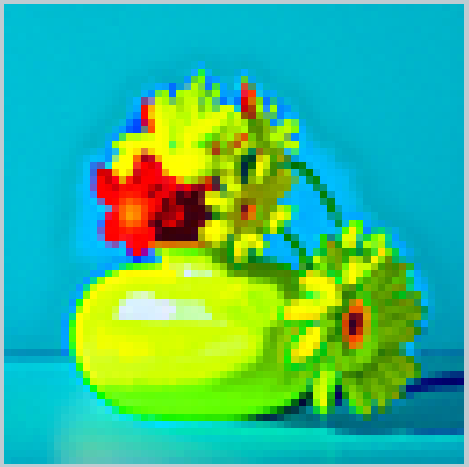
People Dancing



Cat

Highpass prompt: "an oil painting of flowers in a vase"

Lowpass prompt: "a photo of a rubber duck"



Flowers

Rubber Duck

Flowers

Rubber Duck

That's the end of the first part of the project! I thought it was really fun to influence the denoising process of a powerful pre-trained diffusion model, allowing for creativity from both myself and the model.

Project 5b: Diffusion Models from Scratch!

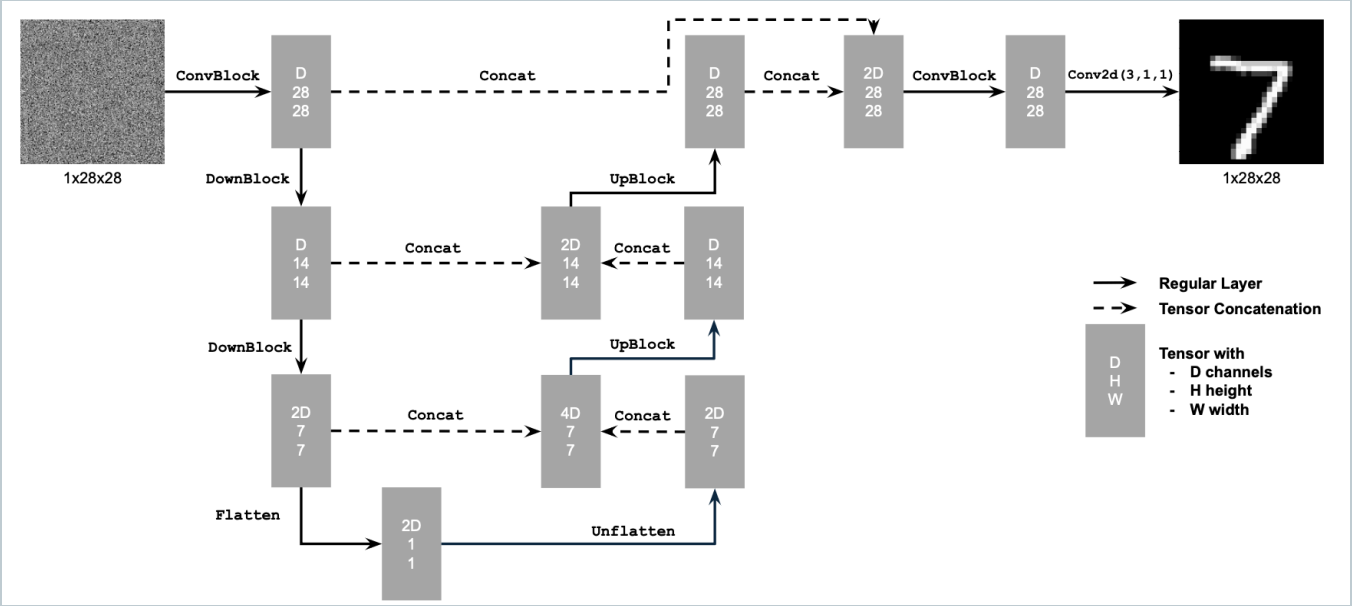
Now we will build our own diffusion models! We unfortunately don't have the resources to train a model as powerful as the DeepFloyd one from the previous part, but we can train a pretty decent one that denoises and generates black-and-white MNIST handwritten digits.

Part 1: Training a Single-Step Denoising UNet

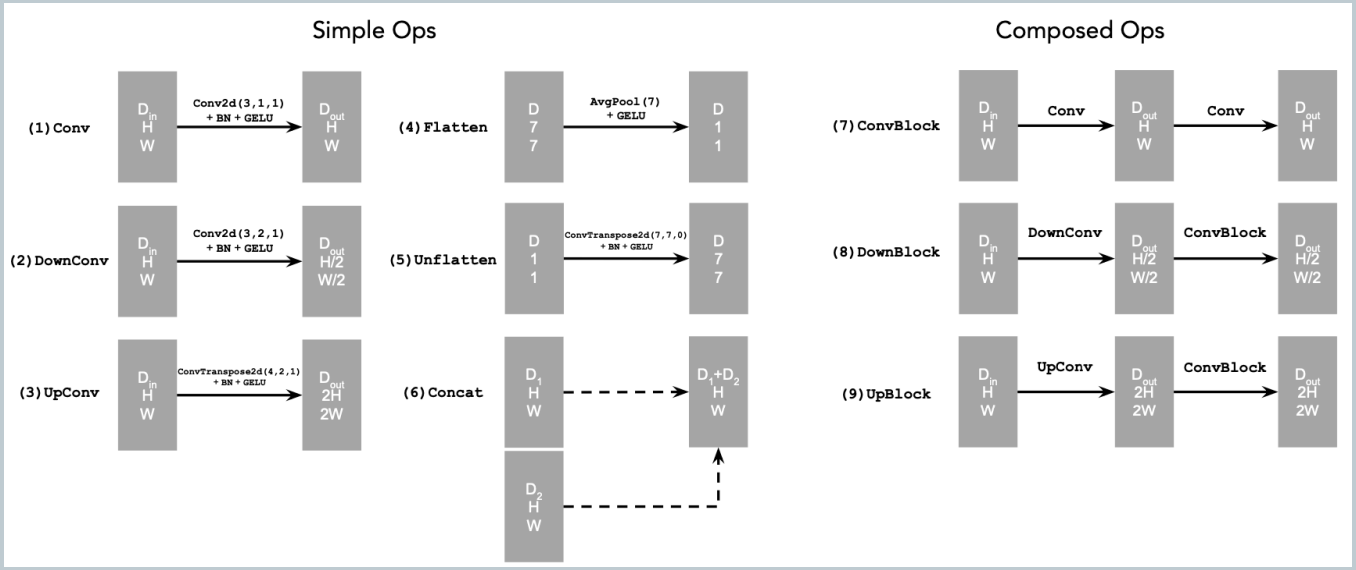
We would like to start by training a simple one-step denoiser.

1.1 Implementing the UNet

We construct the primary architecture of our model, with the help of the pytorch library. Our UNet has the following architecture, and uses a simple L2 loss:



UNet Architecture



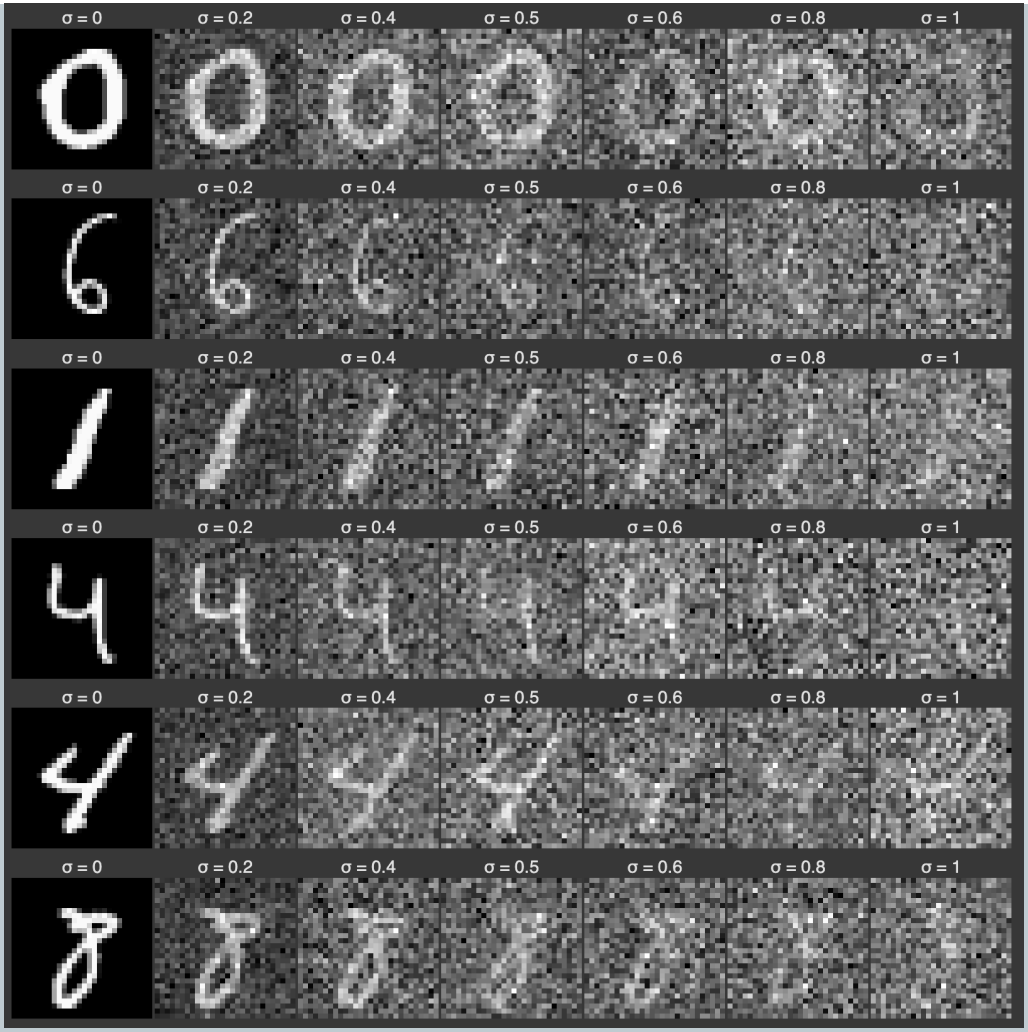
Fine-grained operations in UNet

1.2 Using the UNet to Train a Denoiser

Our first step is to create our training set. We need to manually add noise to the MNIST dataset, and we want to add it with varying degrees of intensity. We use the following equation to do this:

$z = x + \sigma\varepsilon$, where $\varepsilon \sim N(0, I)$.

Here is a sample of our noisy images:

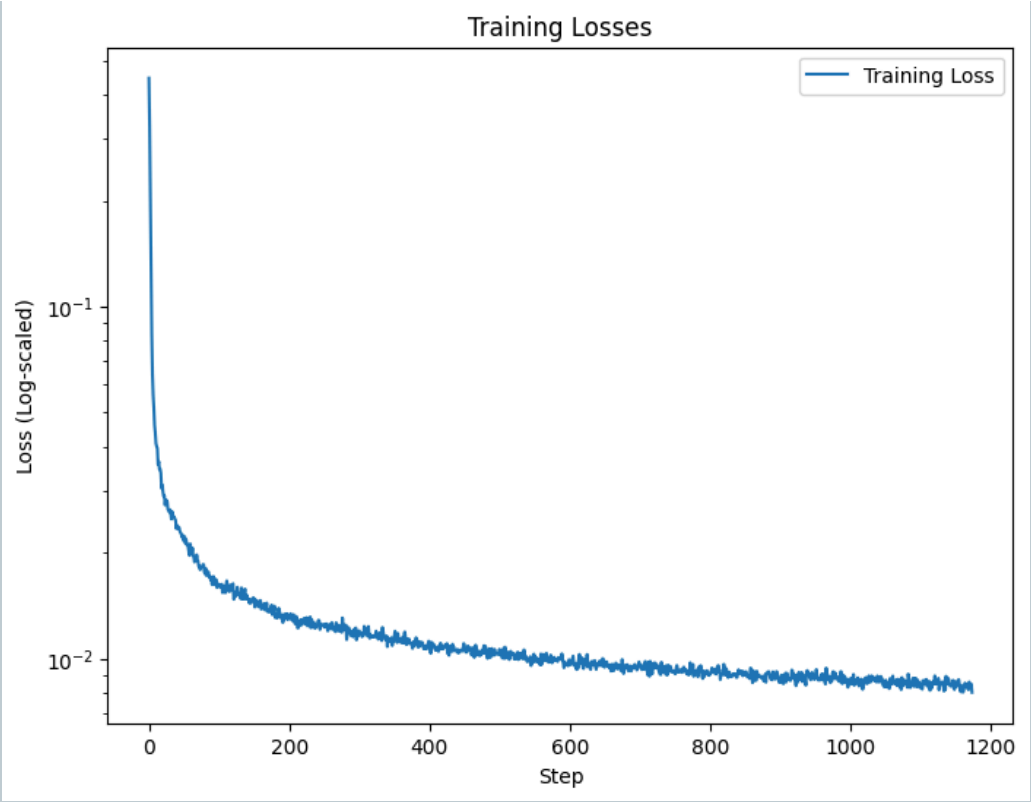


Levels of noise on MNIST digits

1.2.1 Training

We train our UNet to denoise images with noise level $\sigma = 0.5$. We use a batch size of 256, hidden dimension of 128, train for 5 epochs, and use Adam optimizer with a learning rate of 1e-4.

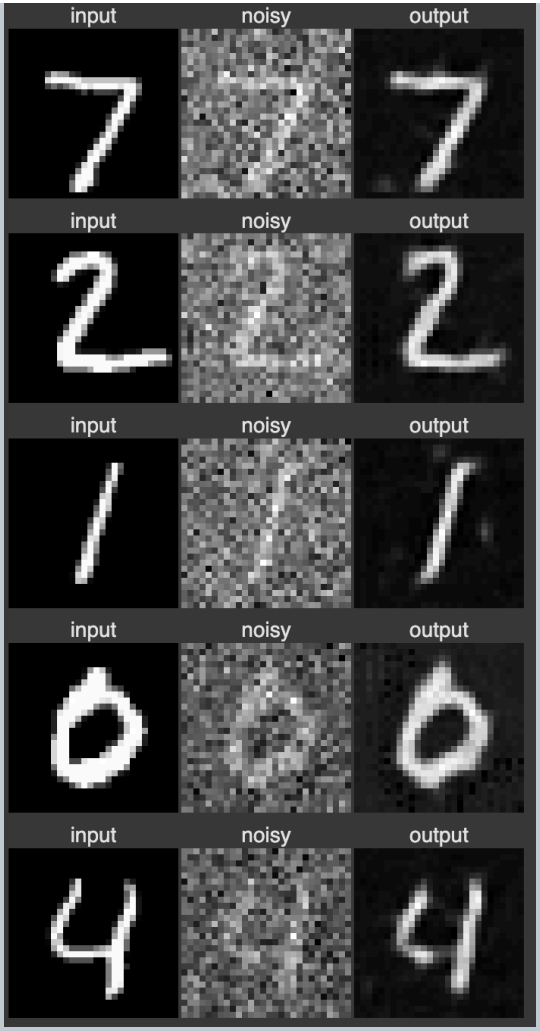
After training, I got the following loss curve:



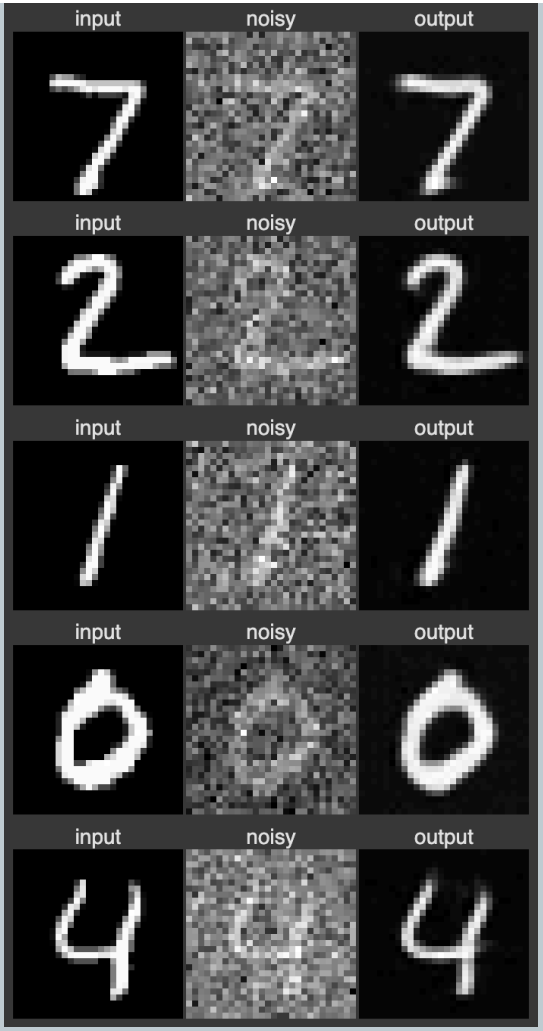
Loss curve

1.2.1 Training

Sampling from the 1st and 5th epochs gives me the following results:



Epoch 1

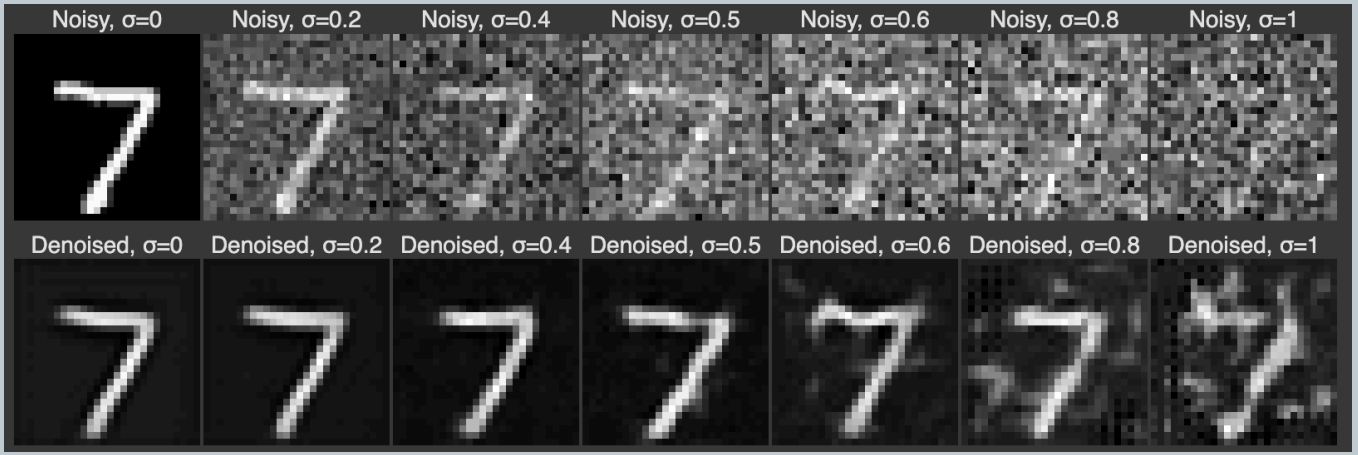


Epoch 5

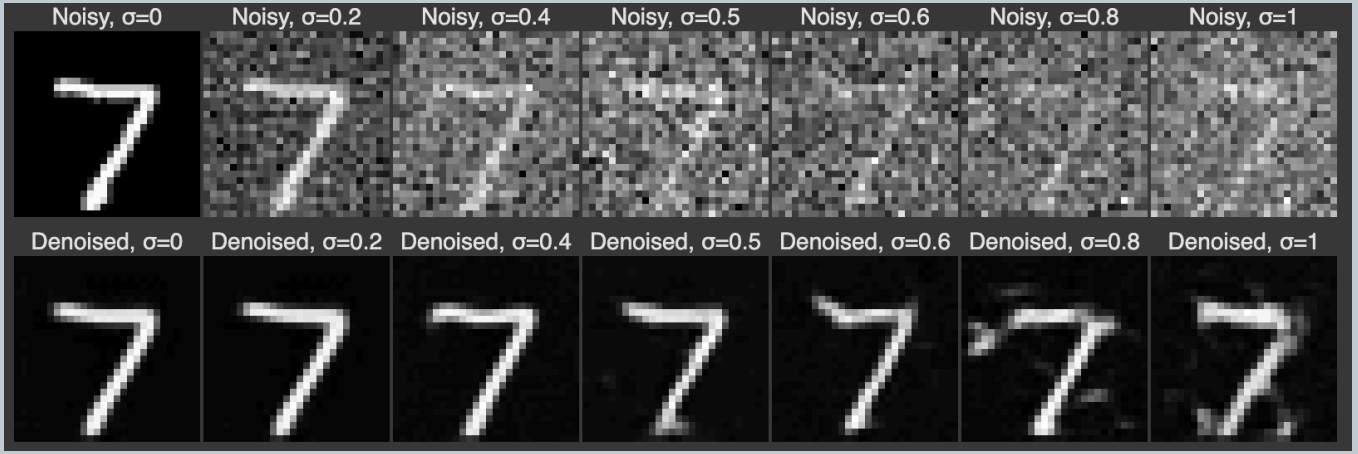
The denoised images at epoch 5 look almost identical to the originals! This tells us our model is performing as desired.

1.2.2 Out-of-Distribution Testing

This model was trained specifically to denoise images with $\sigma = 0.5$. Let's see how it performs on images with other noise levels, like the ones we used to noise our images above:



Epoch 1



Epoch 5

These results are not quite as good. The model still does a decent job at denoising the images at higher noise levels, but it is quite clear that it was not trained for this task.

Part 2: Training a Diffusion Model

Now it's time to try and train diffusion models! We will train a UNet model that can iteratively denoise an image, similar to the process from part A. The goal here is that we can eventually sample images from pure noise using this model, effectively having it generate high quality digits of its own.

There are a few major changes we need to make to our model setup. Firstly, we want our model to predict the noise within an image rather than the clean image itself. We can do this by simply making our loss function the MSE between the predicted and actual noise. We also need to incorporate the timestep into this procedure somehow, as our denoising now depends on the timestep we are at. We use the iterative noising equation from part A to model our image noising:

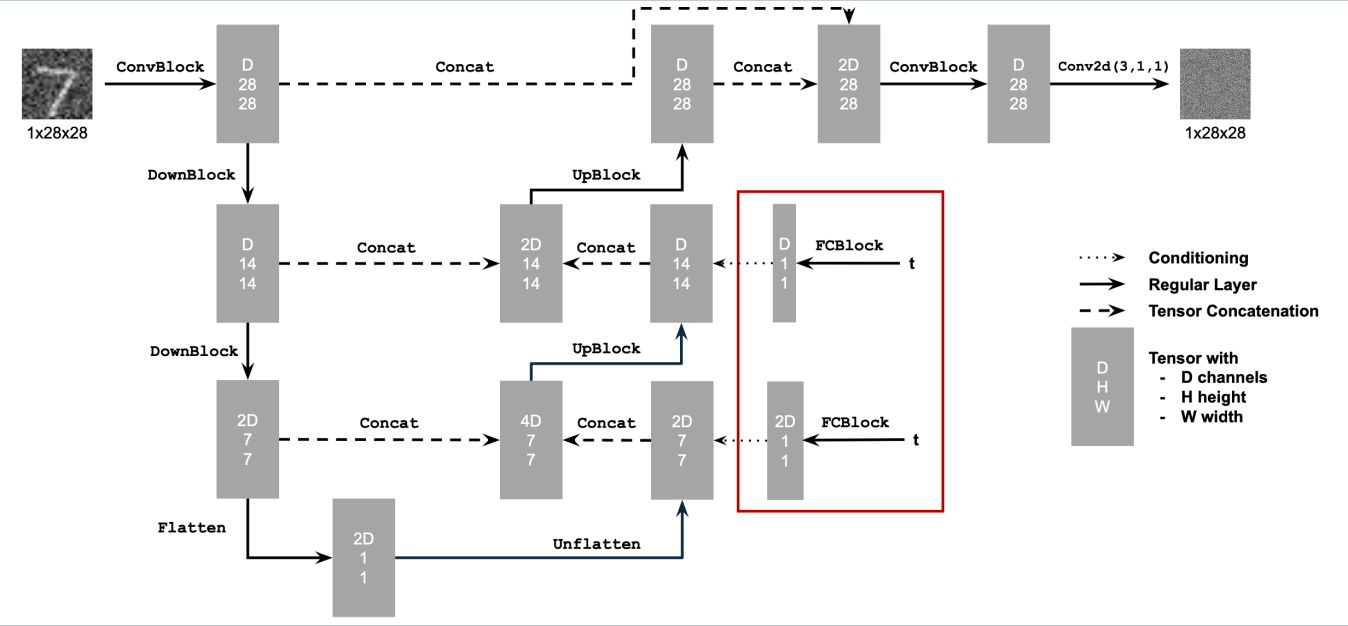
$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \quad \text{where } \epsilon \sim N(0, 1)$$

Along with the following loss function:

$$L = \mathbb{E}_{\epsilon, x_0, t} \| \epsilon_\theta(x_t, t) - \epsilon \|^2$$

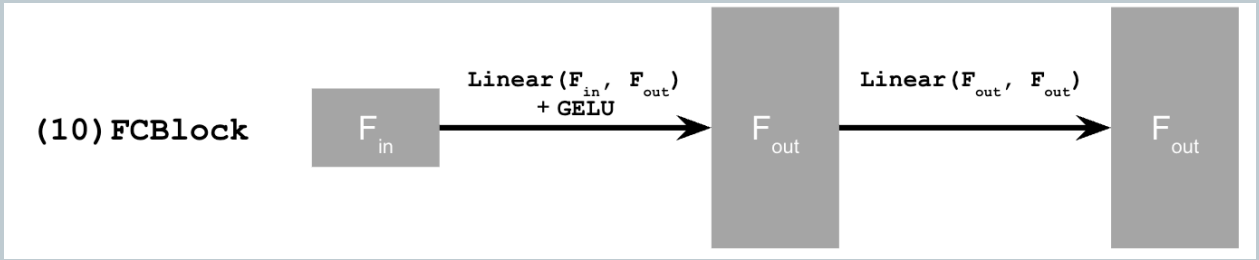
2.1 Adding Time Conditioning to UNet

As stated above, we need to add time-conditioning into our UNet architecture. The suggested way to do so is as follows:



Time-conditioned UNet Architecture

The newly introduced FCBlock in the above architecture looks like this:



FCBlock

2.2 Training the UNet

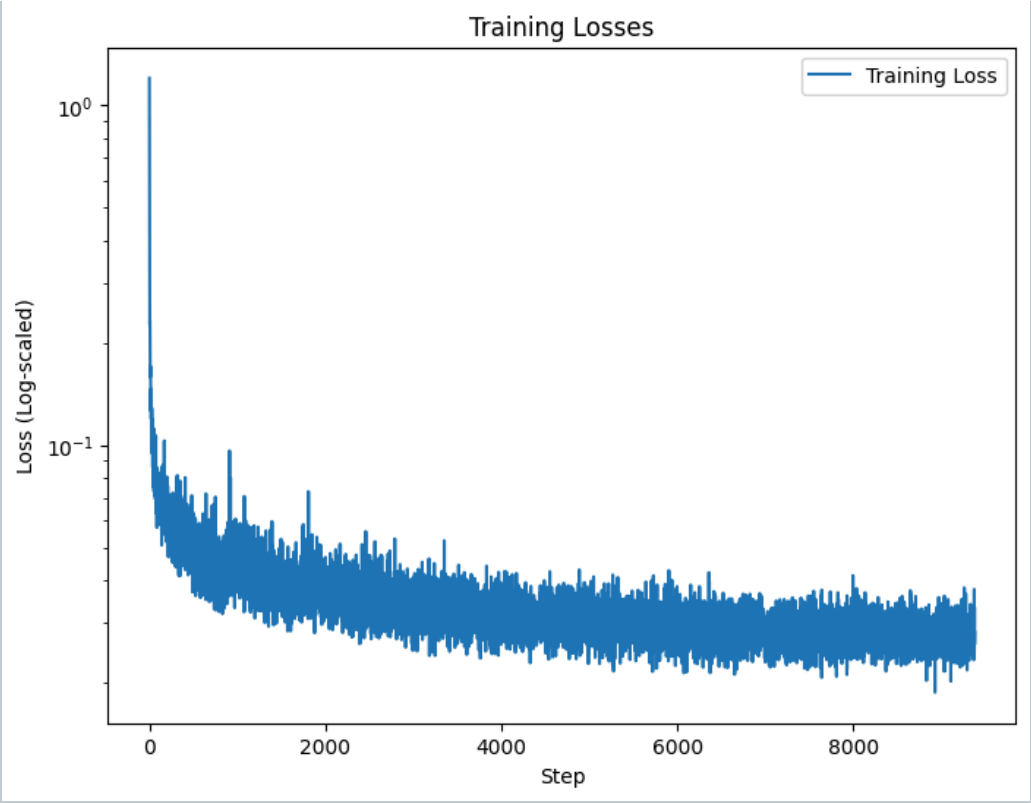
Now we train the UNet using the following algorithm:

Algorithm 1 Training

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

Training Algorithm

We train using batch size 128 and hidden dimension 64 for 20 epochs, and Adam optimizer with initial learning rate of 1e-3 and gamma of 0.1^(1.0/num_epochs). We get the following loss curve:



Loss curve

2.3 Sampling from the UNet

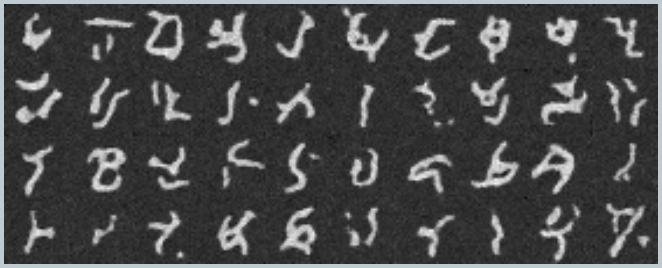
We have our trained UNet, so now we can sample from it. To do this, we can use a very similar algorithm to the one we used in part A:

Algorithm 2 Sampling

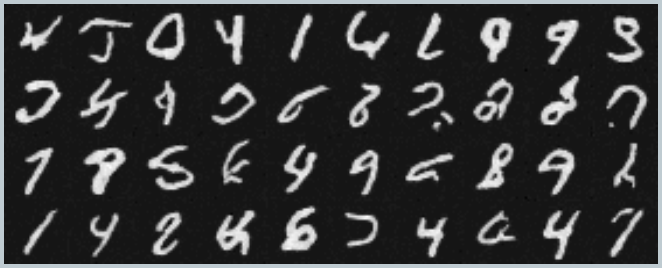
1: Precompute β, α, and $\bar{\alpha}$
2: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
3: **for** t from T to 1, step size -1 **do**
4: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
5: $\quad \hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t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

Sampling Algorithm

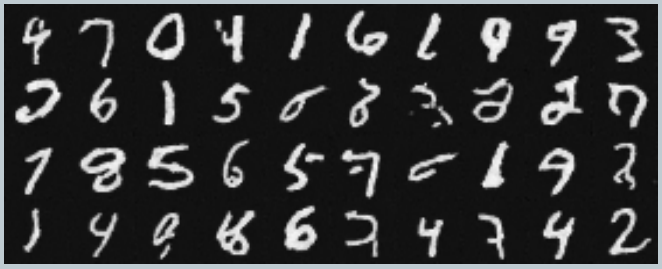
Here are samples from epochs 1, 5, 10, 15, and 20. Please hover your mouse over the image to play the denoising gif.

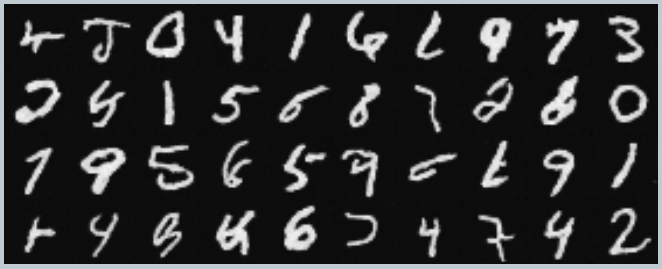


Epoch 1

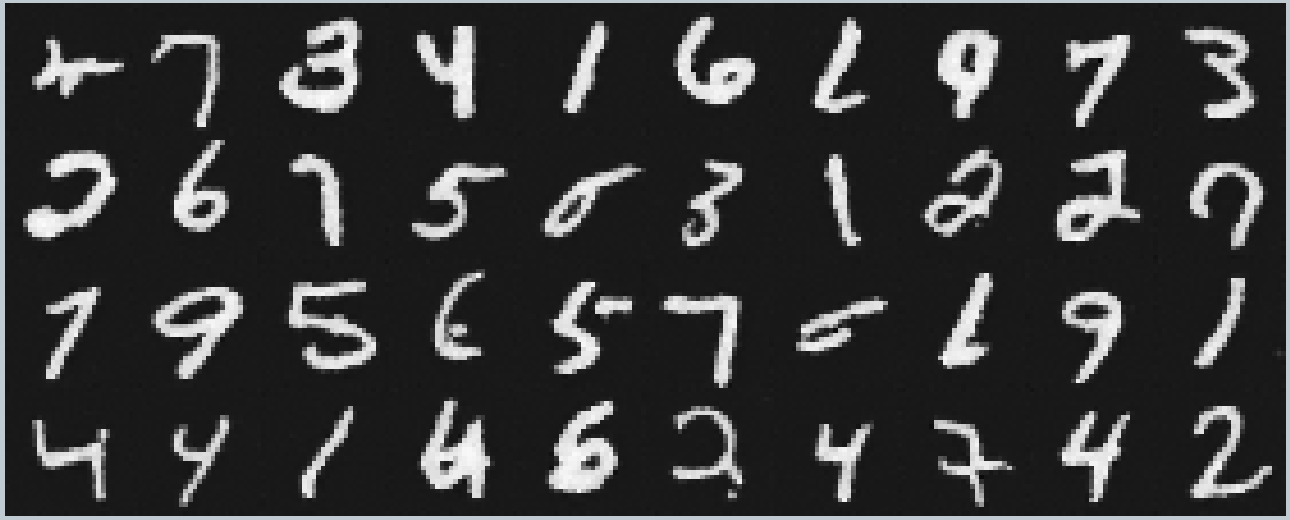


Epoch 5





Epoch 10 Epoch 15



Epoch 20

2.4 Adding Class-Conditioning to UNet

Clearly, the digits produced above are mostly not real numerical digits. We also currently have no way to ask the model to output specific digits that we want. We will now add class-conditioning on top of the existing time-conditioned UNet in order to give us this control for image generation.

To do this, we one-hot encode the image labels during training, pass it through 2 separate FCBlocks as defined above, and then multiply the same unflattening and upsampling steps (that we are already conditioning with t) by the outputs we get.

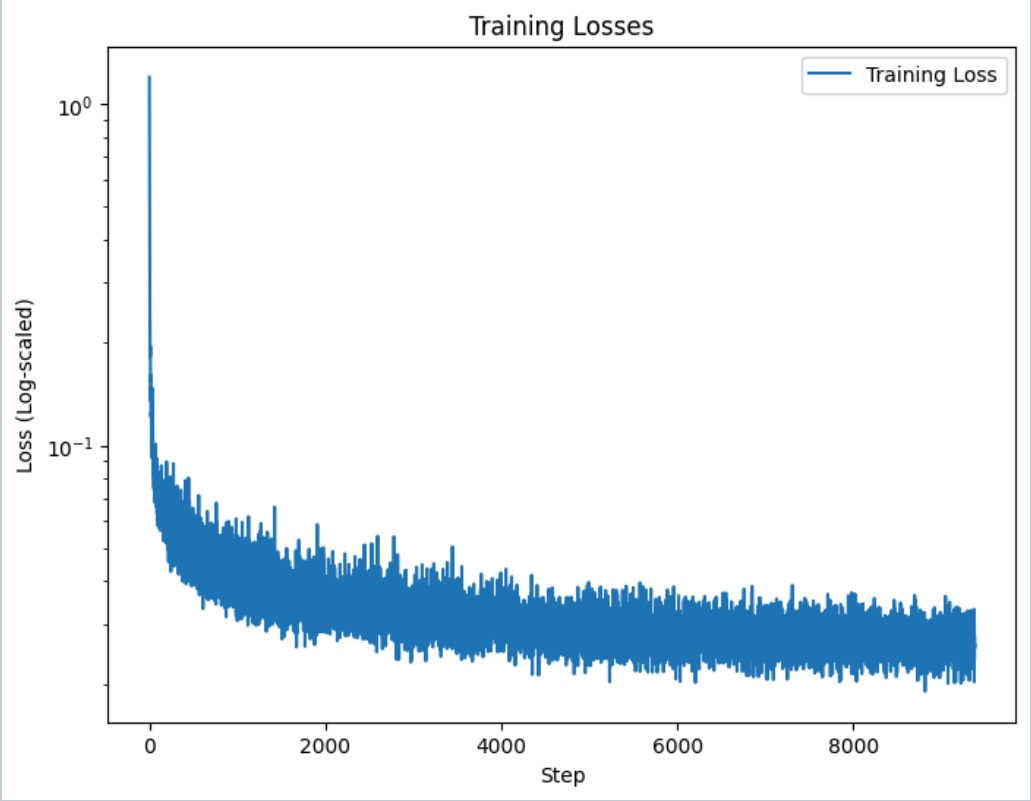
We will also incorporate dropout here, where we set the one-hot encoded vector to all zeros 10% of the time, so that the model still learns to generate unconditionally. Here is the algorithm we use for training:

Algorithm 3 Class-Conditioned Training

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

Training Algorithm

We train using the same parameters as the previous model. We get the following loss curve:



Loss curve

2.5 Sampling from the Class-Conditioned UNet

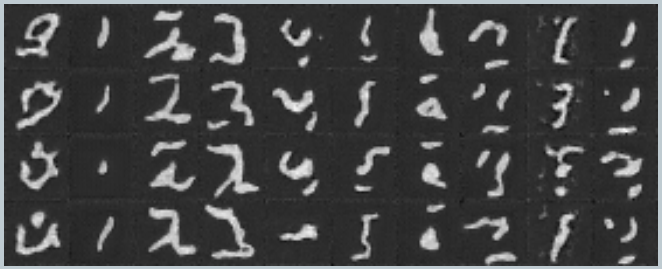
Now we can sample from this model, using a very similar algorithm to the one we used for the time-conditioned model. Here, we have the addition of γ, which we will set to 5.

Algorithm 2 Sampling

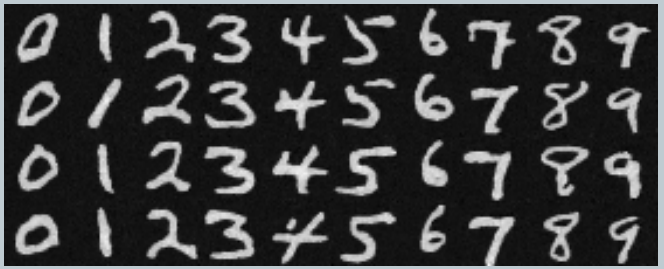
1: Precompute β, α, and $\bar{\alpha}$
2: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
3: **for** t from T to 1, step size -1 **do**
4: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
5: $\quad \hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t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

Sampling Algorithm

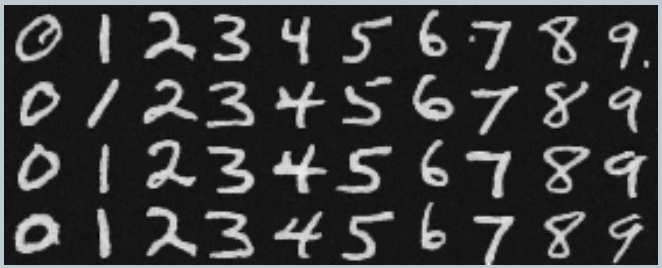
Here are samples from epochs 1, 5, 10, 15, and 20. Please hover your mouse over the image to play the denoising gif.



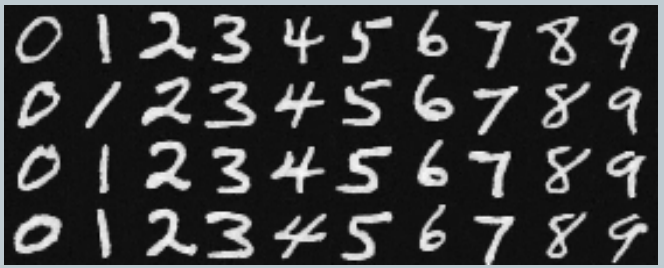
Epoch 1



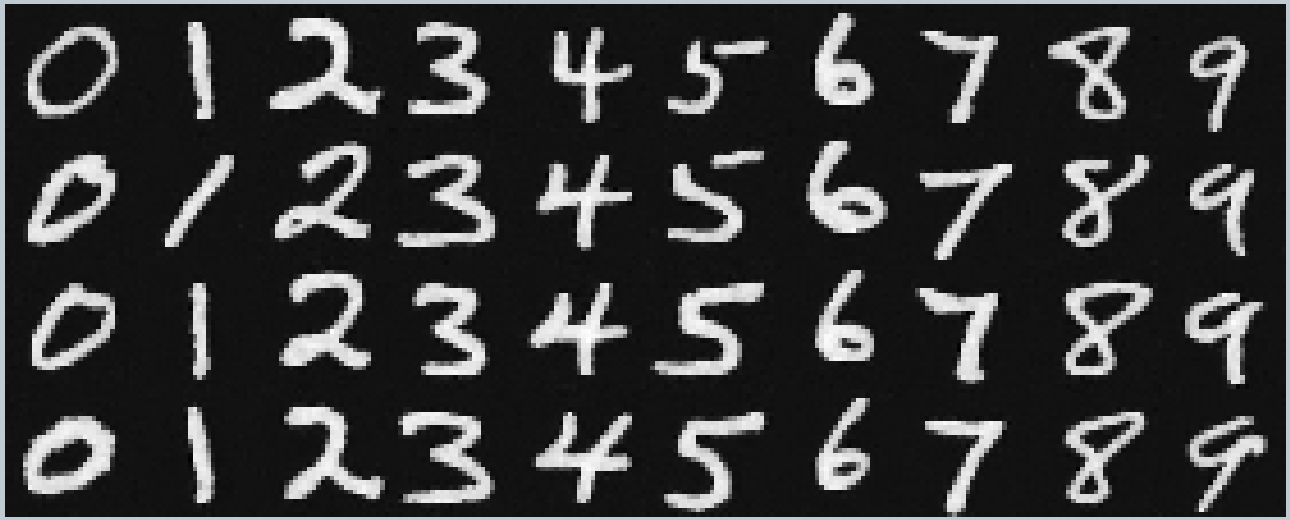
Epoch 5



Epoch 10



Epoch 15



Epoch 20

Looking good! We have finally reached the end of the project. I think this project was the most fascinating one we have built this semester. With the rise of multimodal models these days that generate and understand images perfectly, the whole thing seems like magic. It's cool to finally understand how these models work and to get my hands dirty with an actual, simple implementation from scratch.