Fun With Diffusion Models!

Part A: The Power of Diffusion Models!

In part A, I explore diffusion models, implement sampling loops, and apply them to tasks like inpainting and creating optical illusions.

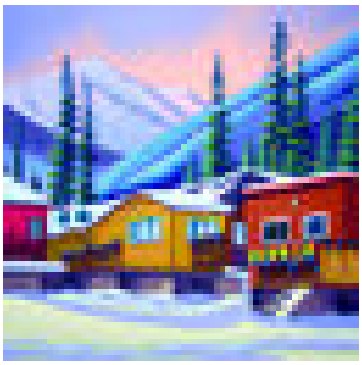
Part 0: Setup

For this part, I instantiate DeepFloyd's `stage_1` and `stage_2` objects used for generation, as well as several text prompts for sample generation. To ensure that the generated images closely align with the textual descriptions, I experimented with various parameter settings, particularly adjusting num_inference_steps to observe changes in output quality. These trials helped me understand the model's ability to control image detail and refinement.

The random seed that I'm using here is 42, and I would use the same seed all subsequent parts.

The text prompts used in this part are: *an oil painting of a snowy mountain village*, *a man wearing a hat* and *a rocket ship*. The corresponding generated images are as below:

Stage 1 with Size [3, 64, 64]

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

We could notice that the generated images are blurred in this stage.

Stage 2 with Size [3, 256, 256]

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

num_inference_steps = 50

an oil painting of a snowy mountain village

a man wearing a hat

a rocket ship

num_inference_steps = 100

an oil painting of a snowy mountain village

a man wearing a hat

a rocket ship

num_inference_steps = 125

We could observe that the quality of the outputs would be higher, i.e. the outputs would be fancier, if we provide text prompts with more details. Increasing the value of `num_inference_steps` would also contribute to the quality of outputs, though it slows down generation. With higher `num_inference_steps` values, the outputs show clearer structure and improved detail generally.

Part 1: Sampling Loops

In this section of the problem set, I create my own "sampling loops" using the pretrained DeepFloyd denoisers to generate high-quality images. I adapt these sampling loops for various tasks, such as inpainting or creating optical illusions.

1.1 Implementing the Forward Process

In this part, I implemente the forward process of the diffusion model, which involves gradually adding noise to a clean image. The forward process is defined by:

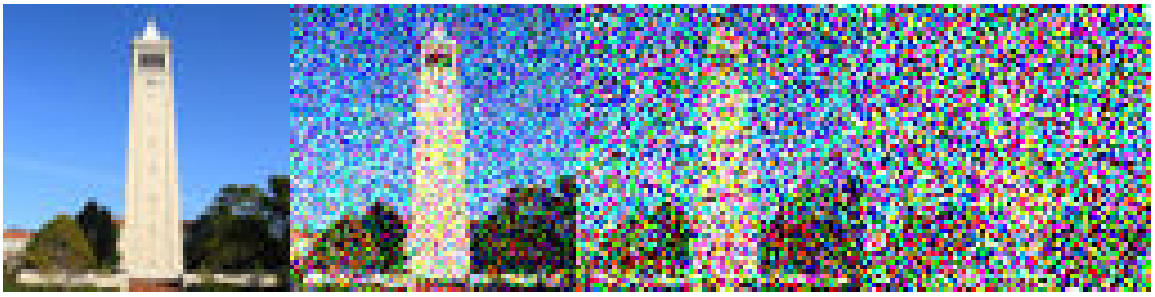
$$q(x_t|x_0) = \mathcal{N}(X_t, \sqrt{\overline{\alpha}}x_0, (1-\overline{\alpha}_t)\mathbf{I}),$$

which is equivalent to computing

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

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 $\sqrt{\overline{\alpha}_t}x_0$ and variance $(1-\overline{\alpha}_t)$.

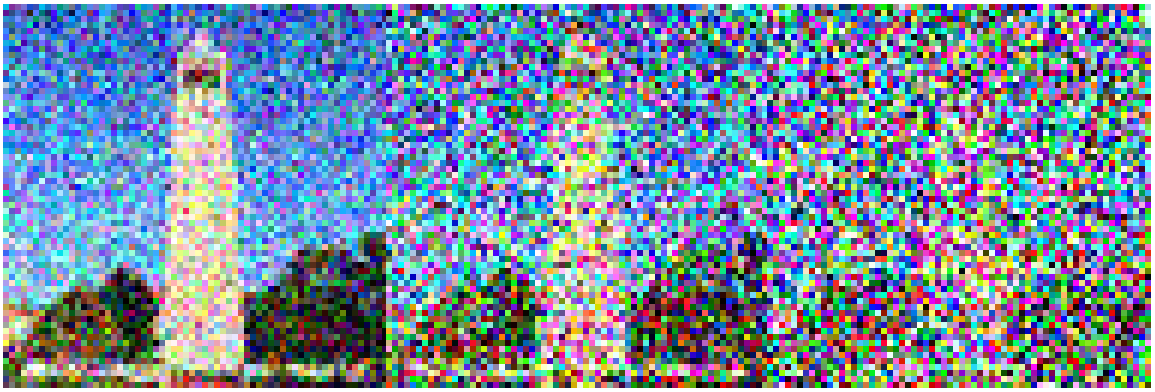
Here is an example of adding noise to `campanile.jpg`:



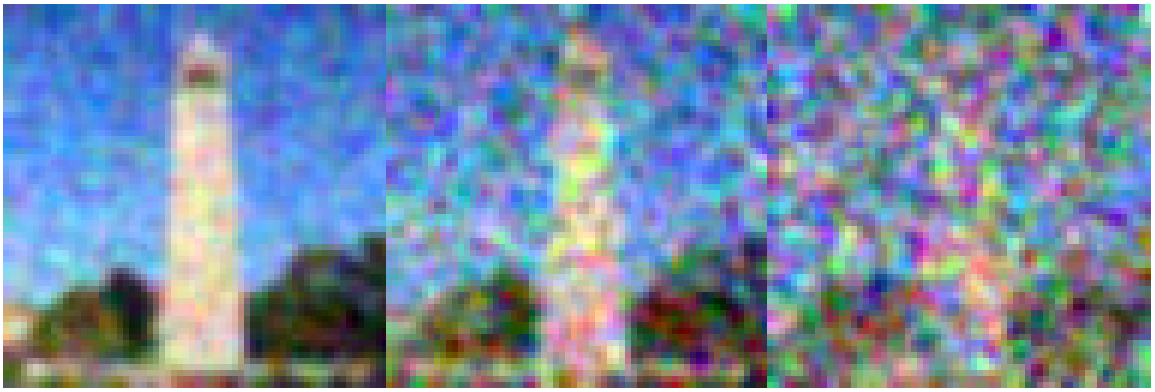
campanile.png **noise level = 250** **noise level = 500** **noise level = 750**

1.2 Classical Denoising

First try to denoise these images using classical methods. Again I work with the noisy images from timesteps $[250, 500, 750]$, applying *Gaussian blur filtering* in an effort to reduce the noise. The results are as below:



Noisy Campanile at t=250 **Noisy Campanile at t=500** **Noisy Campanile at t=750**



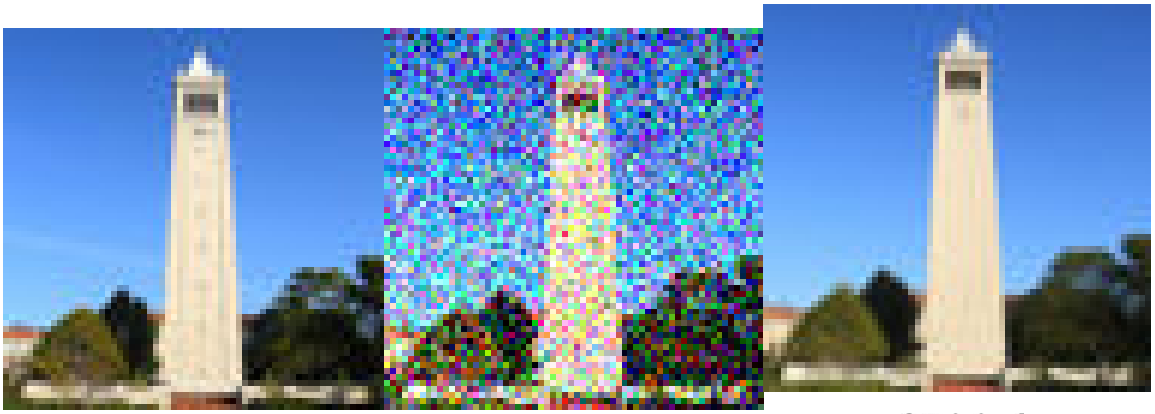
Gaussian Blur Denoising at t=250 **Gaussian Blur Denoising at t=500** **Gaussian Blur Denoising at t=750**

1.3 One-Step Denoising

Now I utilize a pretrained diffusion model to perform denoising. The denoiser is implemented in `stage_1.unet`, which is a UNet architecture that has been extensively trained on a vast dataset of (x_0, x_t) image pairs. This model enables us to estimate the Gaussian noise present in the image, which we can then subtract to retrieve an approximation of the original image.

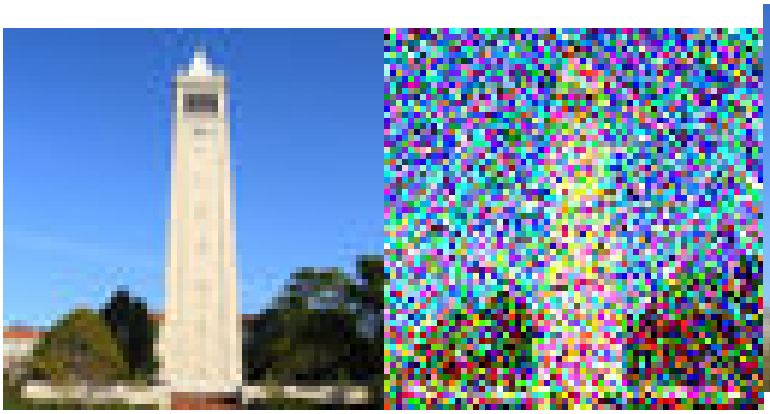
Additionally, the diffusion model requires a text prompt embedding to guide the denoising process. I use `"a high quality photo"` as the relevant text prompt for conditioning the model.

Timestep = 250

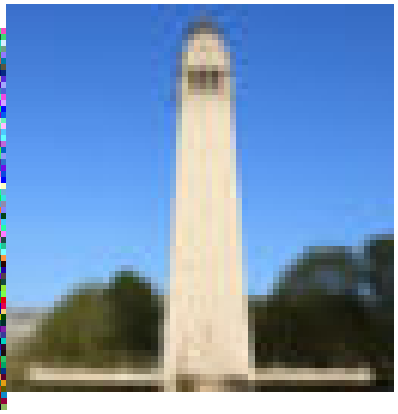


Original Campanile **Noisy Campanile** **Estimate of Original Campanile**

Timestep = 500

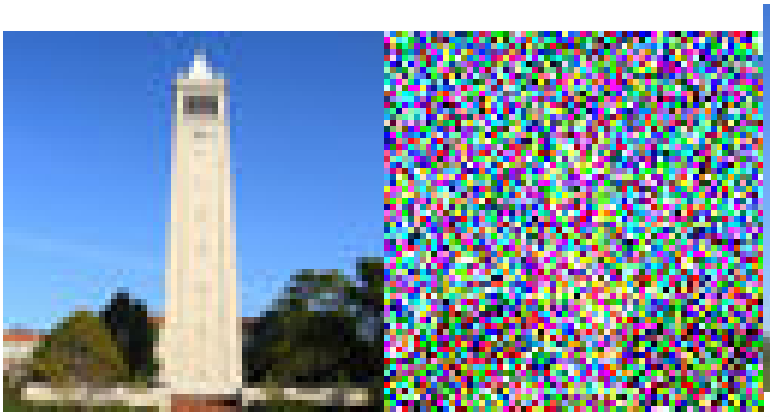


Original Campanile | **Noisy Campanile**



Estimate of Original Campanile

Timestep = 750



Original Campanile | **Noisy Campanile**

Estimate of Original Campanile

1.4 Iterative Denoising

In Part 1.3, we could observe that the denoising UNet performs well at projecting the image onto the natural image manifold, though it worsens as more noise is added. This makes sense since the problem becomes increasingly challenging with higher noise levels.

Diffusion models are designed for iterative denoising. To speed up this process, we can create a new list of timesteps called `strided_timesteps`, allowing us to skip certain steps. The first element in `strided_timesteps` corresponds to the noisiest image , i.e. with the largest timestep, and `strided_timesteps[-1]` corresponds to a clean image. One straightforward way to construct this list is by introducing a regular stride. Here we apply a stride of 30.

On the i-th denoising step, we're at `strided_timesteps[i]` and aim to reach `strided_timesteps[i+1]`, moving from a noisier to a less noisy image. To do this, we apply the following formula:

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

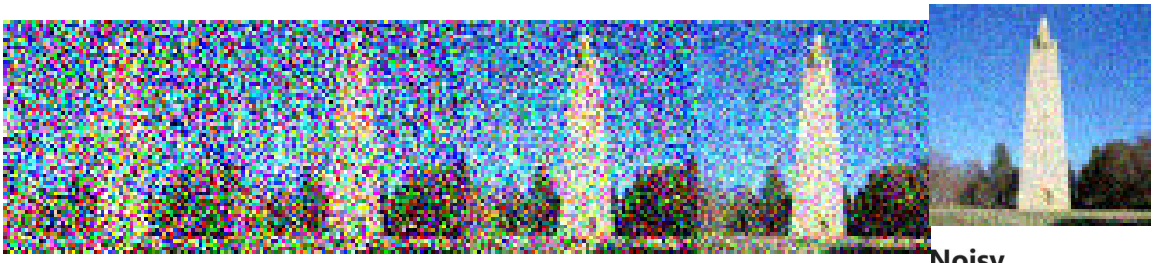
where:

- x_t is the image at timestep $t
- $x_{t'}$ is the noisy image at timestep t' where $t' < t$ (less noisy)
- $\overline{\alpha}_t$ is defined by `alpha_cumprod`
- $\alpha_t = \frac{\overline{\alpha}_t}{\overline{\alpha}_{t'}}$
- $\beta_t = 1 - \alpha_t$
- x_0 is the current estimate of the clean image

This formula gives the current estimate of the clean image, and it's similar to the approach in section 1.3.

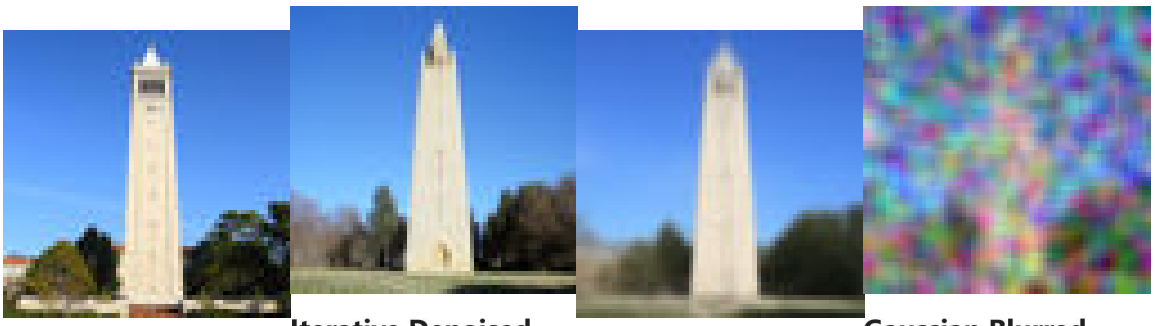
The v_σ is random noise, which in the case of DeepFloyd is also predicted. The function called `add_variance` could add the correct amount of noise to the image.

The noisy images generated in the iteration of denoising are as below:



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**

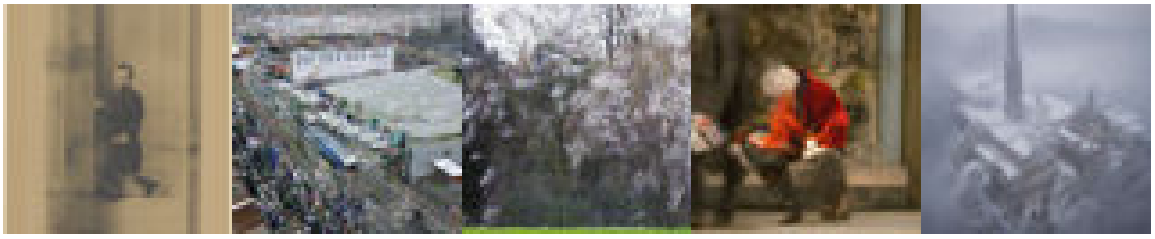
Comparing the result of iterative denoising with the results of the methods before, we could find that both the predicted clean image using iterative denoising and the predicted clean image using only a single denoising step look good, though the iterative method performs better on some details and provides a clearer and fancier image.



Original **Iterative Denoised Campanile** **One-Step Denoised** **Gaussian Blurred Campanile**

1.5 Diffusion Model Sampling

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



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

1.6 Classifier-Free Guidance (CFG)

We could notice that the images generated in the previous section are not of high quality, with some appearing completely nonsensical. To significantly enhance image quality, we can employ a technique known as **Classifier-Free Guidance (CFG)**.

In CFG, we calculate both conditional and unconditional noise estimates, denoted as ϵ_c and ϵ_u. Our new noise estimate is then formulated as:

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

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.

Here are 5 images of `"a high quality photo"` with a CFG scale of $\gamma = 7$, which look much better than those in the prior section:

Sample 1 with CFG **Sample 2 with CFG** **Sample 3 with CFG** **Sample 4 with CFG** **Sample 5 with CFG**

1.7 Image-to-image Translation

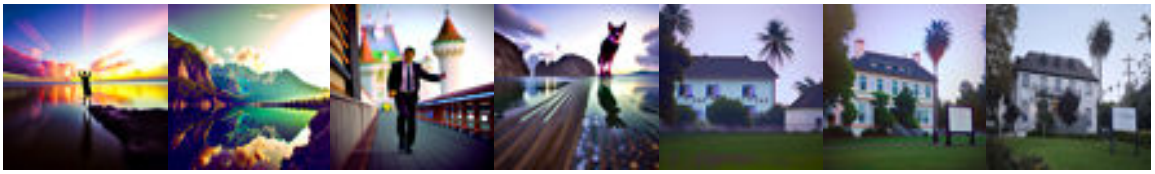
In Part 1.4, we take a real image, add noise to it, and then apply a denoising process. This approach effectively allows us to make modifications to existing images. The more noise we introduce, the greater the potential for edits. This works because, in order to denoise an image, the diffusion model must "imagine" or "hallucinate" some new content—it needs to be "creative." Another way to think about it is that the denoising process "pushes" a noisy image back to the natural image manifold.

&esmp; Here, we'll take the original test image, add a small amount of noise, and then push it back to the image manifold without any conditioning. This will result in an image that resembles the test image (assuming a low-enough noise level). This follows the principles of the **SDEdit algorithm**.

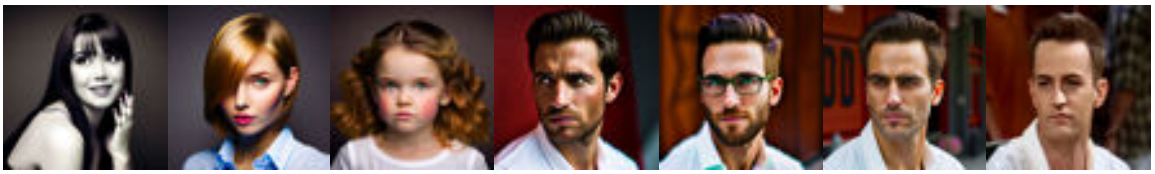
To begin, we will run the forward process to generate a noisy test image, and then use the `iterative_denoise_cfg` function with starting indices of `[1, 3, 5, 7, 10, 20]` steps to display the results, labeling each with its starting index. Here is an example of applying SDEdit to the image of Campanile:

SDEdit with i_start=1	SDEdit with i_start=3	SDEdit with i_start=5	SDEdit with i_start=7	SDEdit with i_start=10	SDEdit with i_start=20	Campanile

I also test on my own images:



SDEdit with i_start=1	SDEdit with i_start=3	SDEdit with i_start=5	SDEdit with i_start=7	SDEdit with i_start=10	SDEdit with i_start=20	House

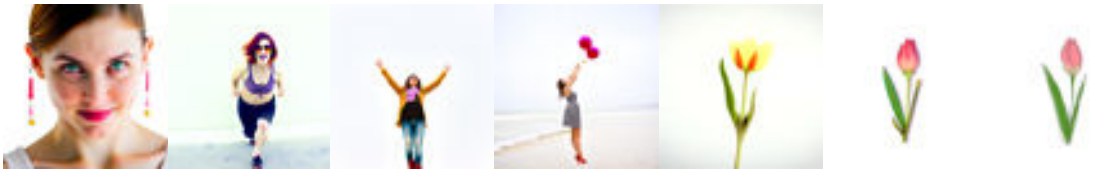


SDEdit with i_start=1	SDEdit with i_start=3	SDEdit with i_start=5	SDEdit with i_start=7	SDEdit with i_start=10	SDEdit with i_start=20	Chandler

1.7.1 Editing Hand-Drawn and Web Images

This approach is especially effective when we start with non-realistic images, such as paintings, sketches, or scribbles, and transform them onto the manifold of natural images. I will try using hand-drawn images to see how they can be creatively mapped onto the natural image manifold.

This is an example of processing a fancy hand-drawn image I downloaded from the web:



Flower at i_start=1	Flower at i_start=3	Flower at i_start=5	Flower at i_start=7	Flower at i_start=10	Flower at i_start=20	Flower

I also create my own works:

Cat at i_start=1	Cat at i_start=3	Cat at i_start=5	Cat at i_start=7	Cat at i_start=10	Cat at i_start=20	Cat

House at i_start=1	House at i_start=3	House at i_start=5	House at i_start=7	House at i_start=10	House at i_start=20	House

1.7.2 Inpainting

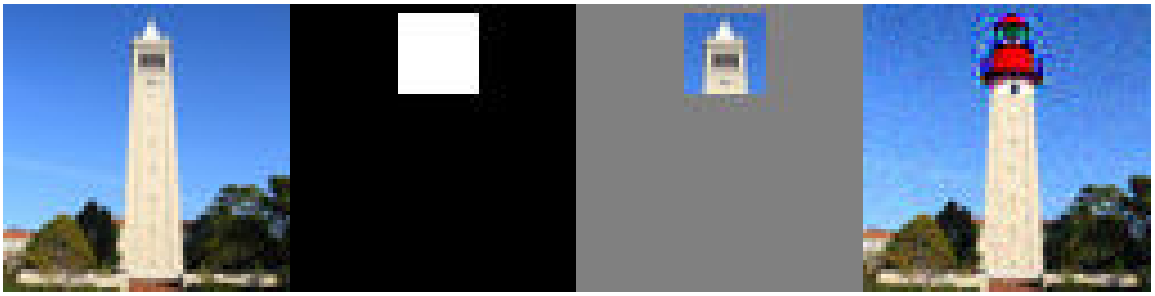
Given an original image x_{orig} and a binary mask $\mathbf{m}$, we can generate a new image that retains the original content where $\mathbf{m} = 0$, while generating new content in the regions where $\mathbf{m} = 1$.

To achieve this, we use the diffusion denoising loop. At each step, after obtaining x_t, we "force" x_t to match the original image in regions where $\mathbf{m} = 0$. This can be expressed as follows:

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

In essence, this approach leaves everything within the mask region untouched, while replacing everything outside the mask with the original image content, with the correct level of noise for the current timestep t.

Now we could edit the picture to inpaint the top of the Campanile:

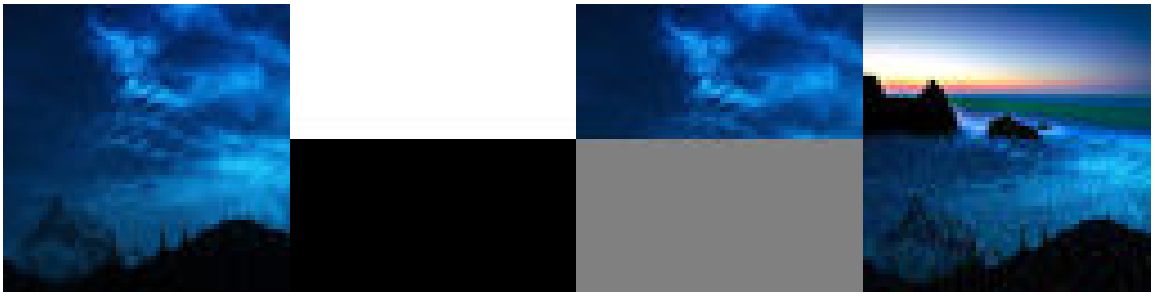


Campanile	Mask	Hole to Fill	Campanile Inpainted

In this case, we could find that the Campanile becomes a lighthouse!

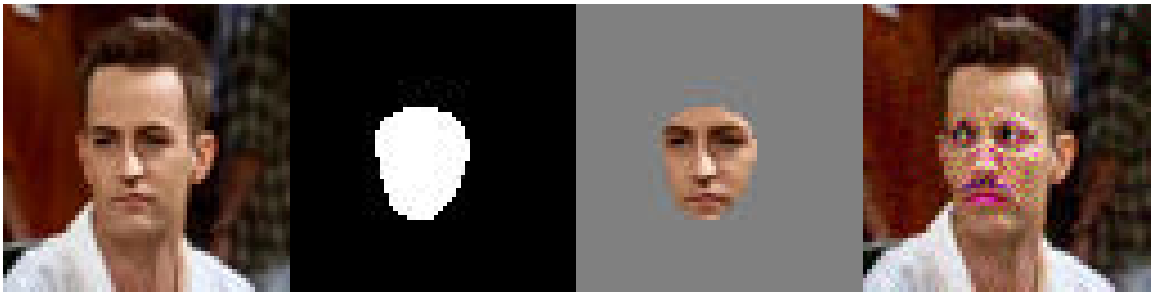
I also apply this to other images:

This is my favourite result. Under the starry sky, the mountains transform into an ocean with islands, a truly dreamy scene.



Mountain **Mask** **Hole to Fill** **Mountain Inpainted**

Here we change the face of Chandler and now he becomes a man with a surprising face.



Chandler **Mask** **Hole to Fill** **Chandler Inpainted**

1.7.3 Text-Conditional Image-to-image Translation

Now, we will replicate the procedure from the previous section, but this time we will guide the projection using a text prompt. This approach goes beyond merely projecting onto the natural image manifold by incorporating language for additional control. All we need to do is replace the prompt `"a high-quality photo"` with one of the precomputed prompts we provide.

Here are some examples:

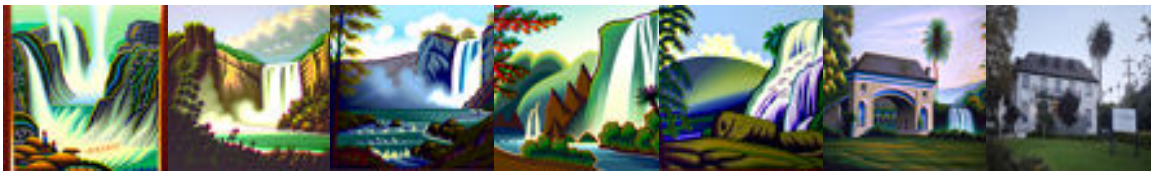
"a rocket ship" on Campanile

Rocket Ship at noise level 1 **Rocket Ship at noise level 3** **Rocket Ship at noise level 5** **Rocket Ship at noise level 7** **Rocket Ship at noise level 10** **Rocket Ship at noise level 20** **Campanile**

"a pencil" on Flower

Pencil at noise level 1 **Pencil at noise level 3** **Pencil at noise level 5** **Pencil at noise level 7** **Pencil at noise level 10** **Pencil at noise level 20** **Flower**

"a lithograph of waterfalls" on House



Waterfall at noise level 1 **Waterfall at noise level 3** **Waterfall at noise level 5** **Waterfall at noise level 7** **Waterfall at noise level 10** **Waterfall at noise level 20** **House**

1.8 Visual Anagrams

In this section, we would implement Visual Anagrams and use diffusion models to create optical illusions. For example, our goal is to produce an image that appears as `"an oil painting of people around a campfire"`, but when flipped upside down, reveals `"an oil painting of an old man"`.

To achieve this, we'll denoise the image at step t using the prompt `"an oil painting of an old man"`, obtaining a noise estimate ϵ_1. At the same time, we'll flip the image upside down and denoise it with the prompt `"an oil painting of people around a campfire"`, resulting in noise estimate ϵ_2. By flipping ϵ_2 back to its original orientation and averaging it with ϵ_1, we get a combined noise estimate. Finally, we apply a reverse diffusion step with this averaged noise estimate.

The full algorithm will be:

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

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

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

where UNet is the diffusion model UNet from before, $\mathbf{flip}(\cdot)$ is a function that flips the image, and p_1 and p_2 are two different text prompt embeddings. And our final noise estimate is ϵ.

Here are some examples:

In this image, you can see some people sitting by a campfire, but when it is flipped upside down, it transforms into an old man.

An Oil Painting of People around a Campfire

An Oil Painting of an Old Man

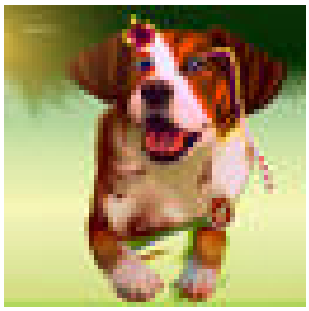
I got this pair of images using `"a photo of the amalfi cost"` and `"an oil painting of a snowy mountain village"`. In the image on the left, you can see the beautiful Amalfi Coast. When flipped, it reveals a snowy mountain with villages afar.

A Photo of the Amalfi Cost

An Oil Painting of a Snowy Mountain Village

These images are generated from `"a photo of a man"` and `"a photo of a dog"`. In this image, there is a self-portrait of a serious man; when flipped, it transforms into a cute Beagle dog.

A Photo of a Man



A Photo of a Dog

1.9 Hybrid Images

In this section, I implement **Factorized Diffusion** to create hybrid images, similar to what we did in Project 2.

To make hybrid images with a diffusion model, we can apply a related approach. I generate a composite noise estimate by using two distinct text prompts to estimate the noise separately. Then, combine the low frequencies from one noise estimate with the high frequencies from the other. The process is as follows:

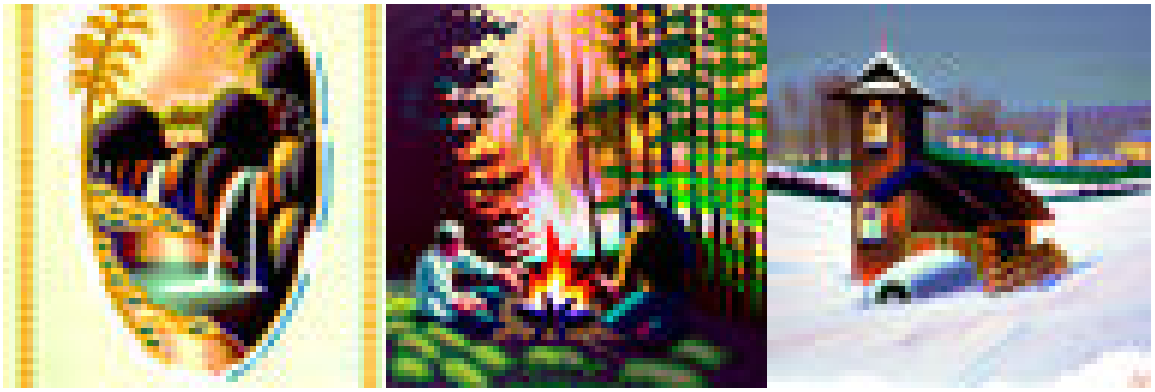
$$\epsilon_1 = \mathbf{UNet}(x_t, t, p_1)$$

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

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

where UNet is the diffusion model UNet, f_{lowpass} is a low pass function, f_{highpass} is a high pass function, and p_1 and p_2 are two different text prompt embeddings. Our final noise estimate is ϵ. For the low-pass and high-pass filters, I simply apply a Gaussian blur with a kernel size of 33 and a sigma of 2.

Here are some of my examples. We could see waterfalls, two man sitting by the campfire and a snowy village in the images below.



A Lithograph of Waterfalls **An Oil Painting of People Around a Campfire** **An Oil Painting of a Snowy Mountain Village**

Scale them down to simulate the scene as if viewed from a distance. Now we could find a skull, an old man with white beards and a dog.

A Lithograph of a Skull

An Oil Painting of an Old Man

A Photo of a Dog

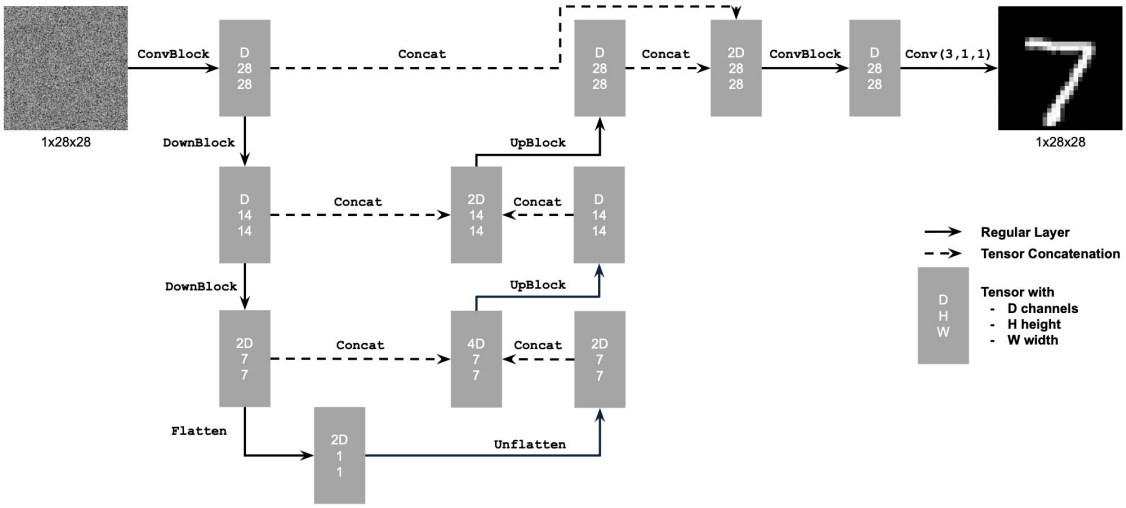
Part B: Diffusion Models from Scratch!

In this part, I will train my own diffussion model on MNIST.

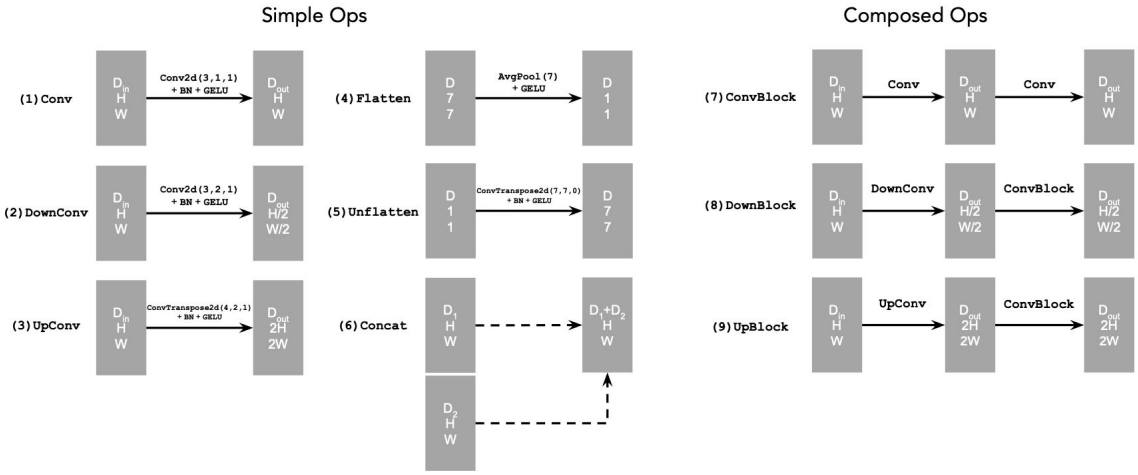
Part 1: Training a Single-Step Denoising U-Net

1.1 Implementing the UNet

In this part, we will implement a denoiser as a U-Net. It consists of a few downsampling and upsampling blocks with skip connections. First we define a few tensor operations:



The diagram above uses a number of standard tensor operations defined as follows:



where:

- **Conv2d(kernel_size, stride, padding)** is `nn.Conv2d()`.
- **BN** is `nn.BatchNorm2d()`.
- **GELU** is `nn.GELU()`.
- **ConvTranspose2d(kernel_size, stride, padding)** is `nn.ConvTranspose2d()`.
- **AvgPool(kernel_size)** is `nn.AvgPool2d()`.

At a high level, the blocks do the following:

- **(1) Conv** is a convolutional layer that doesn't change the image resolution, only the channel dimension.
- **(2) DownConv** is a convolutional layer that downsamples the tensor by 2.
- **(3) UpConv** is a convolutional layer that upsamples the tensor by 2.
- **(4) Flatten** is an average pooling layer that flattens a 7×7 tensor into a 1×1 tensor. 7 is the resulting height and width after the downsampling operations.
- **(5) Unflatten** is a convolutional layer that unflattens/upsamples a 1×1 tensor into a 7×7 tensor.
- **(6) Concat** is a channel-wise concatenation between tensors with the same 2D shape. This is simply `torch.cat`.
- `D` is the number of hidden channels and is a hyperparameter that we will set ourselves.

We define composed operations using our simple operations in order to make our network deeper. This doesn't change the tensor's height, width, or number of channels, but simply adds more learnable parameters.

- **(7) ConvBlock**, is similar to **Conv** but includes an additional **Conv**. Note that it has the same input and output shape as **(1) Conv.**
- **(8) DownBlock**, is similar to **DownConv** but includes an additional **ConvBlock**. Note that it has the same input and output shape as **(2) DownConv.**
- **(9) UpBlock**, is similar to **UpConv** but includes an additional **ConvBlock**. Note that it has the same input and output shape as **(3) UpConv.**

1.2 Using the UNet to Train a Denoiser

For this part, we aim to solve the following denoising problem: Given a noise z, we aim to train a denoiser D_θ such that it maps the noise to a clean image x. To do so, we could

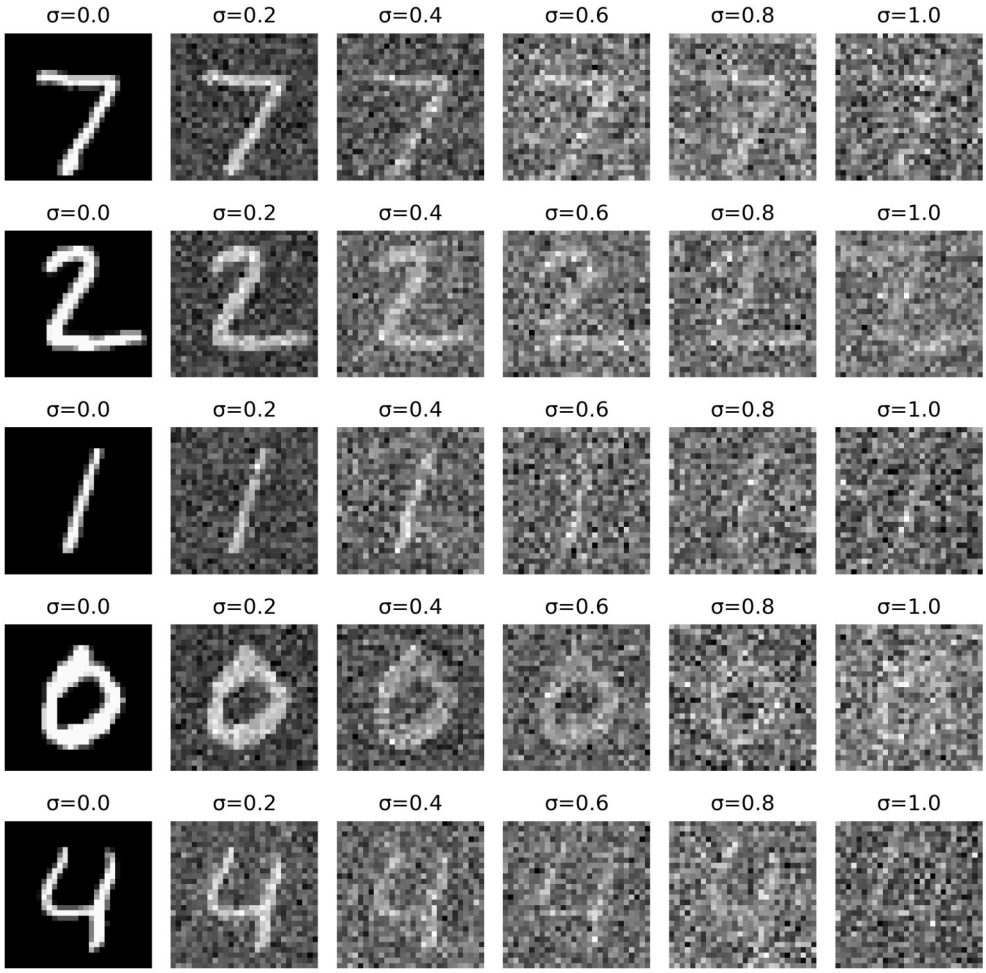
optimize over an L2 loss:

$$L = \mathbb{E}_{z,x} ||D_\theta(z) - x||^2.$$

To train our denoiser, we need to generate training data pairs of (z, x), where each x is a clean MNIST digit. For each training batch, we can generate z from x using the the following noising process:

$$z = x + \sigma\epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, I).$$

Here are some examples for different noising processes over $\sigma = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]$:

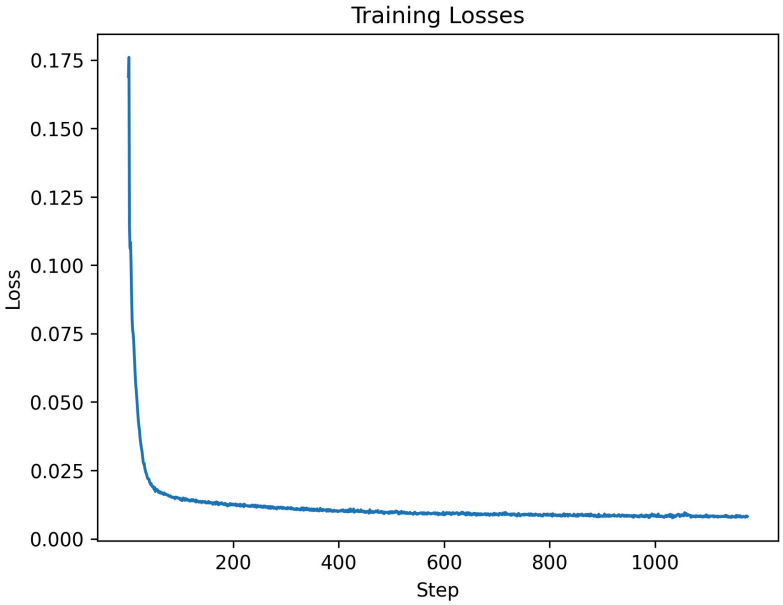


1.2.1 Training

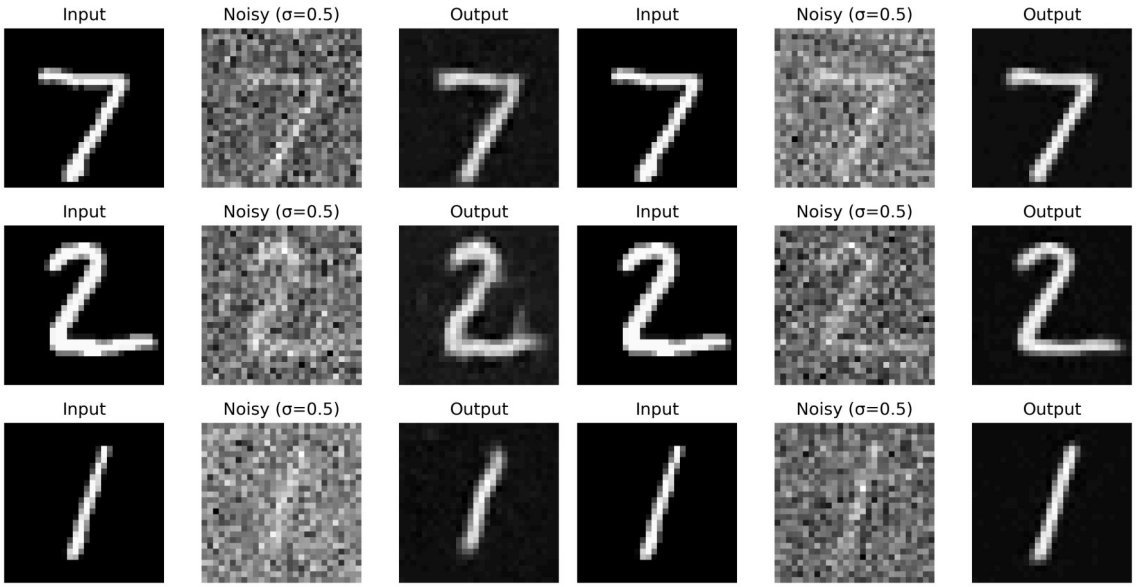
Now I will train the model to perform denoising. Here I just set $\sigma = 0.5$ when training. I use Adam as the optimizer and use the following hyperparameters:

```
num_hidden = 128
    batch_size = 256
    num_epochs = 5
    lr = 1e-4
```

Here is a training loss curve ploting every few iterations during the whole training process:



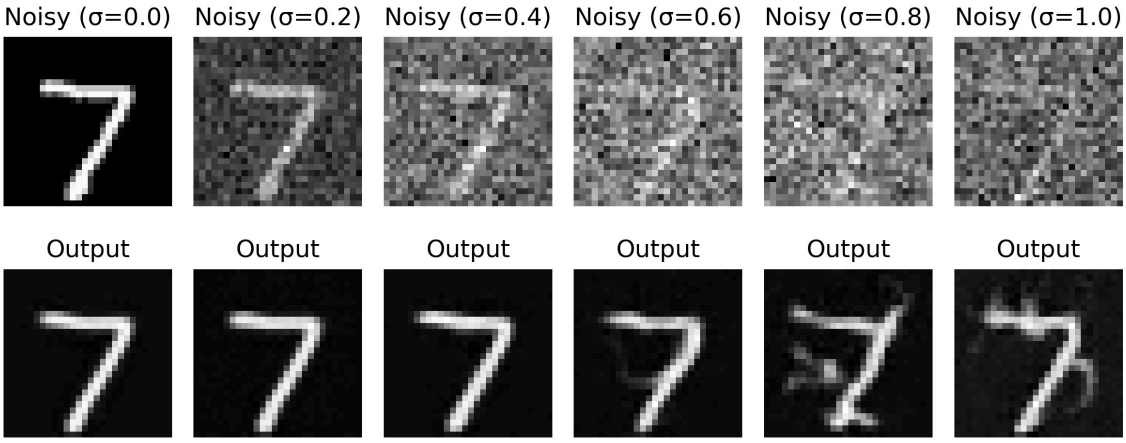
The sample results after the 1st and 5th epoch are shown as below:



Results after 1 epoch of training **Results after 5 epochs of training**

1.2.2 Out-of-Distribution Testing

Once trained enough, we could use our denoising UNet on noisy samples from our test set. I test by keeping the same image, and varying $\sigma = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]$. The results on the test set are as below:



Part 2: Training a Diffusion Model

We are now prepared to start the diffusion process, where we will train a UNet model capable of iteratively denoising images. In this section, we will implement **DDPM (Denoising Diffusion Probabilistic Model)**.

We will first introduce one small difference: we can change our UNet to predict the added noise ϵ instead of the clean image x. Mathematically, these are equivalent since $x = z - \sigma\epsilon$. Therefore, the new loss function becomes:

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

where ϵ_θ is a UNet trained to predict noise.

In diffusion, our ultimate goal is to start with a pure noise image and generate a realistic image from it. However, as we observed in *Part 1*, a single-step denoising approach does not produce satisfactory results. Instead, we need to denoise the image *iteratively* to achieve better outcomes.

We could generate noisy images x_t from x_0 for some timestep t for $t \in \{0, 1, \cdots, T\}$:

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

Intuitively, when $t = 0$ we want x_t to be the clean image x_0, when $t = T$ we want x_t to be pure noise ϵ, and for $t \in \{1, \cdots, T-1\}$, x_t should be some linear combination of the two. Here is a DDPM recipe to build a list $\overline{\alpha}$ for $t \in \{0, 1, \cdots, T\}$ utilizing lists α and β:

- Create a list β of length T such that $\beta_0 = 0.0001$ and $\beta_T = 0.02$ and all other elements β_t for $t \in \{1, \cdots, T-1\}$ are evenly spaced between the two. Here β is known as the variance schedule; it controls the amount of noise added at each timestep.
- $\alpha_t = 1 - \beta_t$.
- $\overline{\alpha}_t$ is a cumulative product of α_s for $s \in \{1, \cdots, t\}$.

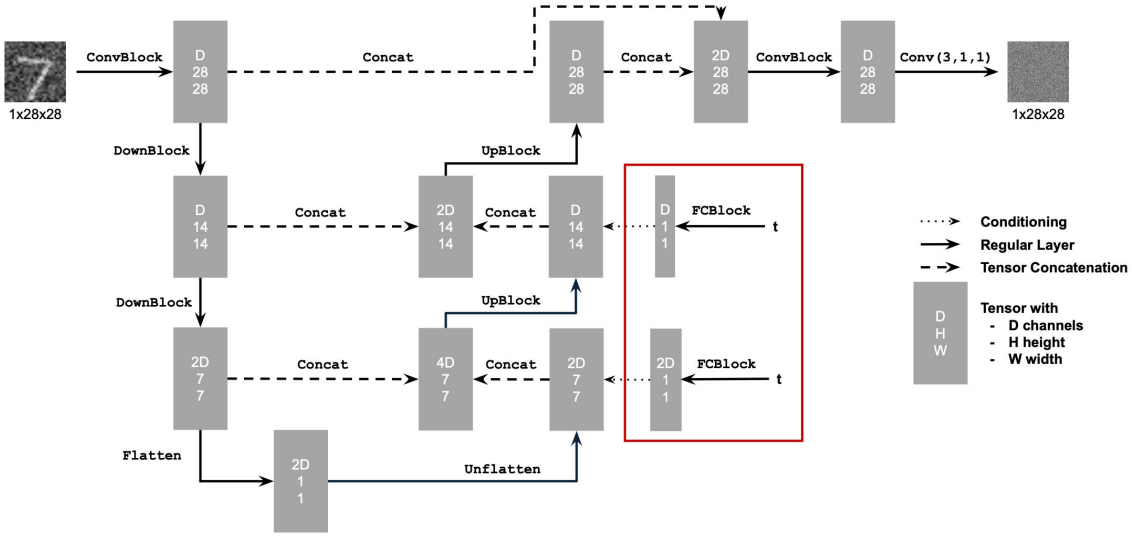
To denoise image x_t, we could simply apply our UNet ϵ_θ on x_t to obtain the noise ϵ. However, this approach would be suboptimal because the UNet expects the noisy image to have a specific noise variance $\sigma = 0.5$ for the best results, while the variance of x_t actually varies with each timestep t. Although training T separate UNets for each timestep is

possible, it's far more efficient to condition a single UNet on the timestep t. This brings us to our final objective:

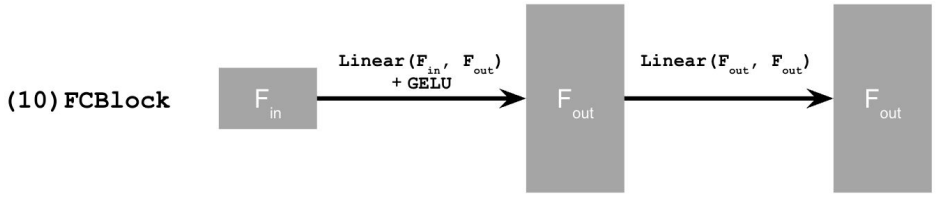
$$L = \mathbb{E}_{\epsilon, x_0, t} ||\epsilon_\theta(x_t, t) - \epsilon||^2.$$

2.1 Adding Time Conditioning to UNet

We need a method to incorporate the scalar as a conditioning element in our UNet model. Here is a possible way:



This approach introduces a new operator, called **FCBlock (fully-connected block)**, which we use to inject the conditioning signal into the UNet:



Here **Linear(F_in, F_out)** is a linear layer with **F_in** input features and **F_out** output features, which could be implemented using `nn.Linear`.

Since the conditioning signal t is a scalar, **F_in** should be of size 1. Here I also normalize t to be in the range $[0, 1]$ before embedding it, i.e. pass in $\frac{t}{T}$. The pseudocode for embedding t is as follows:

```
fc1_t = FCBlock(...)
    fc2_t = FCBlock(...)

    # the t passed in here should be normalized to be in the range [0,
1]
    t1 = fc1_t(t)
    t2 = fc2_t(t)

    # Follow diagram to get unflatten.
    # Replace the original unflatten with modulated unflatten.
    unflatten = unflatten + t1
    # Follow diagram to get up1.
    ...
    # Replace the original up1 with modulated up1.
```

```
        up1 = up1 + t2
        # Follow diagram to get the output.
        ...
```

2.2 Training the UNet

Training our time-conditioned UNet ϵ_θ is straightforward now. We simply select a random image from the training set, choose a random timestep t, and train the denoiser to predict the noise present in the image x_t. This process is repeated with various images x_t and timestep values t until the model converges to satisfactory performance.

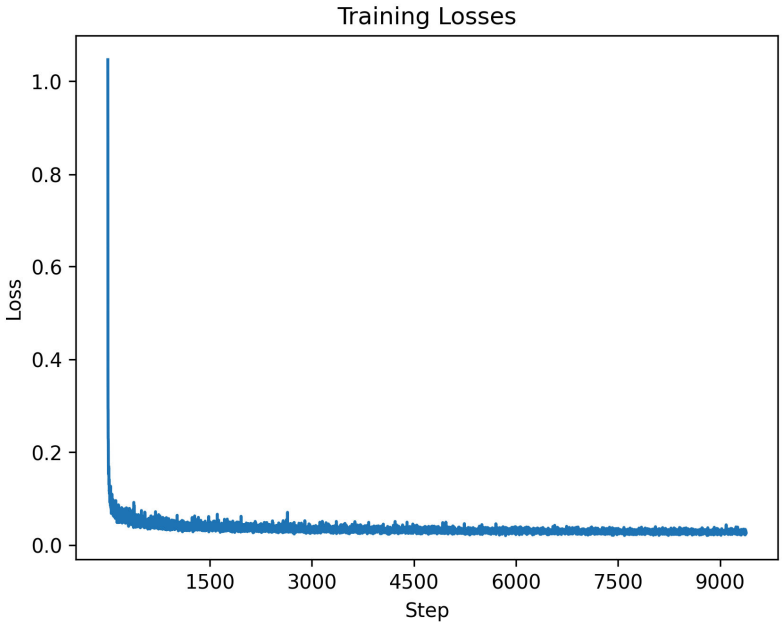
Algorithm 1 Training

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

- **Objective**: I will train a time-conditioned UNet $\epsilon_\theta(x_t, t)$ to predict the noise in x_t given a noisy image x_t and a timestep t.
- **Dataset and dataloader**: I use the MNIST dataset via `torchvision.datasets.MNIST` with flags to access training and test sets with shuffling them before creating the dataloader.
- **Model**: I use the time-conditioned UNet architecture defined in section 2.1.
- **Optimizer**: I use Adam optimizer with an initial learning rate of `1e-3`. I also use an exponential learning rate decay scheduler with a gamma of $0.1^{(1.0/\text{num\_epochs})}$, which can be implemented using `scheduler = torch.optim.lr_scheduler.ExponentialLR(...)`. In addition, I call `scheduler.step()` after every epoch.
- **Hyperparameters**: To be specific, I also use the following hyperparameters:

```
num_hidden = 64
  batch_size = 128
  num_epochs = 20
  lr = 1e-3
```

A training loss curve plot for the time-conditioned UNet over the whole training process is shown as below:



2.3 Sampling from the UNet

The pseudocode for the sampling process is as follows:

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** $\mathbf{x}_0$

Here are some sampling results for the time-conditioned UNet:

Epoch 1 **Epoch 5** **Epoch 10**

Epoch 15 **Epoch 20**

2.4 Adding Class-Conditioning to UNet

To improve results and allow more control over image generation, we can optionally condition our UNet on the digit class 0-9. This will involve adding two additional **FCBlock**s to our UNet. For the class-conditioning vector c, I use a one-hot encoded vector rather than

a single scalar. Since we also want the UNet to work without class conditioning, I apply dropout, setting the class conditioning vector to 0 about 10% of the time ($p_{\text{uncond}} = 0.1$). Here is one approach to condition our UNet $\epsilon_\theta(x_t, t, c)$ on both time t and class c:

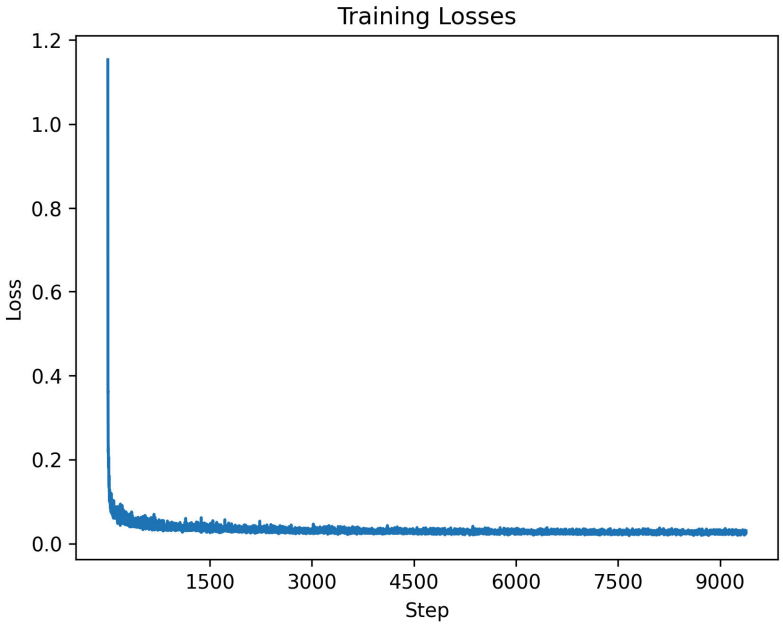
```
fc1_t = FCBlock(...)
    fc1_c = FCBlock(...)
    fc2_t = FCBlock(...)
    fc2_c = FCBlock(...)

    t1 = fc1_t(t)
    c1 = fc1_c(c)
    t2 = fc2_t(t)
    c2 = fc2_c(c)

    # Follow diagram to get unflatten.
    # Replace the original unflatten with modulated unflatten.
    unflatten = c1 * unflatten + t1
    # Follow diagram to get up1.
    ...
    # Replace the original up1 with modulated up1.
    up1 = c2 * up1 + t1
    # Follow diagram to get the output.
    ...
```

Training for this section will be similar to the time-only training, with the main difference being the addition of the conditioning vector c and periodically performing unconditional generation.

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: $t \sim \text{Uniform}(\{1, \dots, T\})$
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
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}} \right\|^2$
11: **until** happy

A training loss curve plot for the class-conditioned UNet over the whole training process is shown as below:



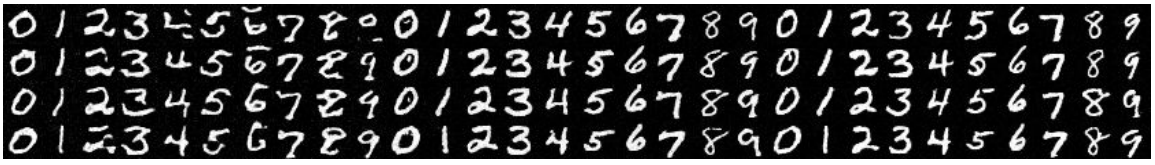
2.5 Sampling from the Class-Conditioned UNet

The pseudocode for the sampling process is as follows:

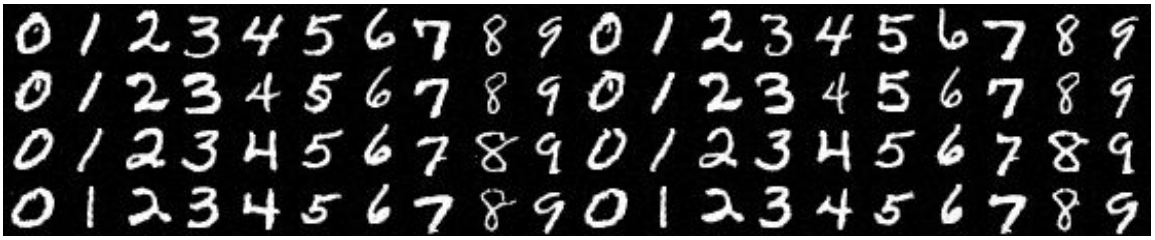
Algorithm 4 Class-Conditioned Sampling

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 $\mathbf{x}_0$

Here are some sampling results for the class-conditioned UNet where I use classifier-free guidance with $\gamma = 5.0$:

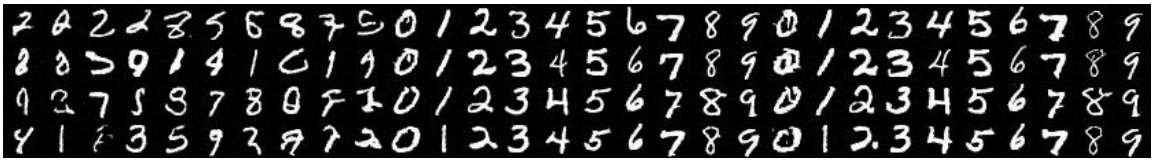


Epoch 1 **Epoch 5** **Epoch 10**



Epoch 15 **Epoch 20**

I was also curious about the influence of guidance scale, so I tried `guidance_scale = [0, 5, 10]`,



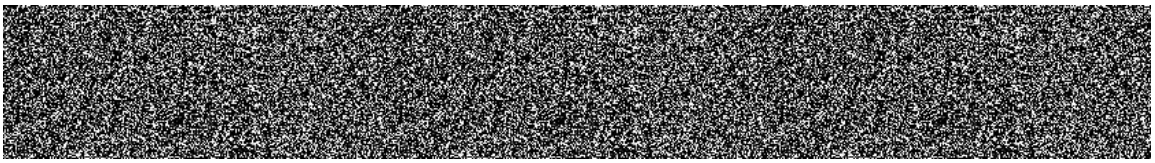
guidance_scale = 0 **guidance_scale = 5** **guidance_scale = 10**

We could find that `guidance_scale = 5` performs best, while `guidance_scale = 0` provides strange digits and `guidance_scale = 10` emphasizes the characteristic features too much.

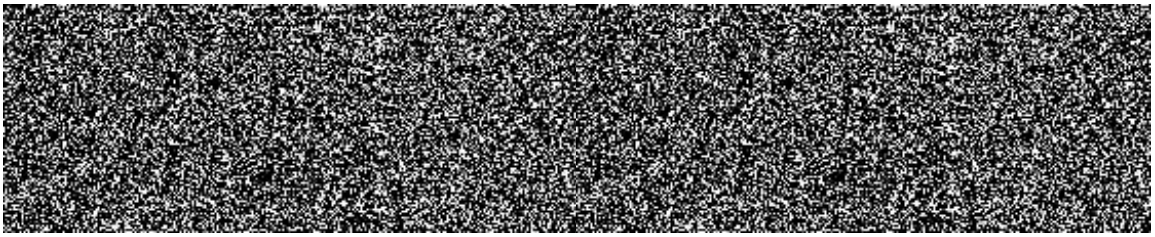
Part 3: Bells & Whistles

- **Sampling Gifs**: I create my own sampling gifs as the course website shown, so please refer to my website:)

  Here are some sampling result gifs for the time-conditioned UNet:

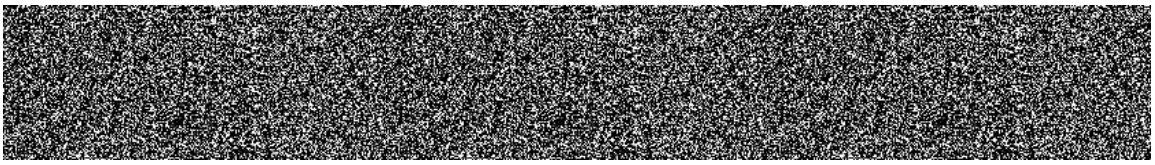


Epoch 1 **Epoch 5** **Epoch 10**

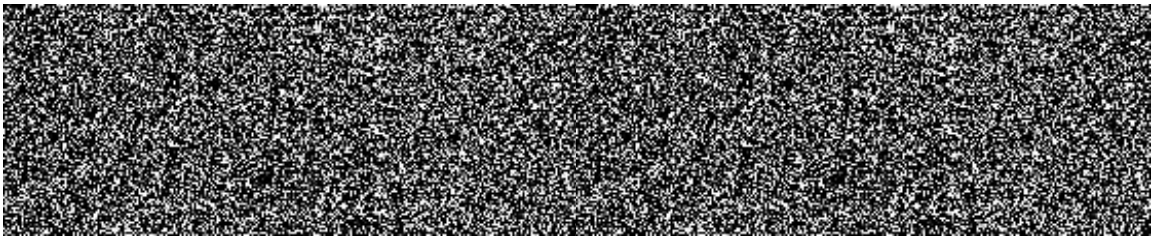


Epoch 15 **Epoch 20**

The sampling result gifs for the class-conditioned UNet with $\gamma = 5.0$ are as follows:



Epoch 1 **Epoch 5** **Epoch 10**



Epoch 15 **Epoch 20**

Cool Stuff I Learnt

In this project, I gained hands-on experience with diffusion models, starting with pre-trained models in Part A, where I explored their functionality by implementing diffusion sampling loops and applying them to tasks such as inpainting and creating optical illusions. This provided me with a solid understanding of how diffusion models work and how they can be utilized for various image generation tasks. In Part B, I furthered my knowledge by training my own diffusion model on the MNIST dataset, which allowed me to understand the nuances of model training, conditioning, and how to generate images from noise. Overall, I really enjoyed the process of learning the principles and application methods from the ground up.