Saatvik Billa CS180 Project 5A: Image Warping and Mosaicing

Part 0: Image Generation with DeepFloyd IF

In the setup part of the project, we loaded in a DeepFloyd IF model that generated the The model first generates 64x64 pixel images which are then upscaled to 256x256 pixels in the second stage, which are the ones depicted.

To test the model's capabilities, I experimented with three prompts: "an oil painting of a snowy mountain village," "a man wearing a hat," and "a rocket ship." Using a fixed random seed of 180 for reproducibility, I ran each stage with both 40 and 20 inference steps. The generation process involved feeding text embeddings through Stage 1 to create initial small images, which Stage 2 then transformed into larger, more detailed versions, which are the ones that you see.

The model demonstrated consistent performance across various types of prompts, from landscapes to portraits to objects, while maintaining efficient resource usage throughout the process.

num_inference=20

num_inference=40

Part 1.1: Implementing the Forward Process

The process follows a formula where a clean image is transformed using a combination of scaling and noise addition. My implementation uses a pre-computed array of alpha values (alphas_cumprod) that determines how much of the original image and noise should be preserved at each timestep.

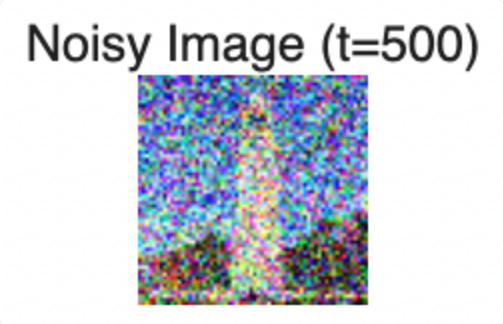
I implemented a forward function that takes two inputs: an image and a timestep t. For each timestep, the function retrieves the corresponding alpha value and generates Gaussian noise using PyTorch's randn_like function. The noisy image is then created by combining two components: the original image scaled by the square root of alpha, and random noise scaled

by the square root of (1 - alpha).

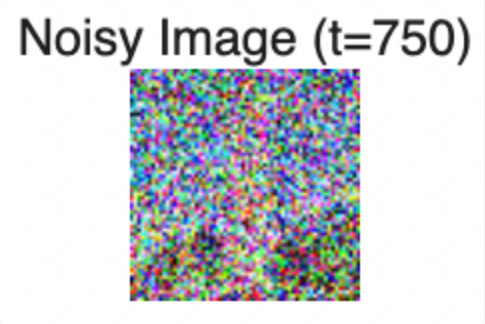
To test the implementation, I applied the forward process to a test image at three different timesteps: 250, 500, and 750. The results demonstrate the progressive nature of the noise addition. At t=250, the image maintains most of its original features with some added noise, while at t=750, the image is heavily degraded by noise, though faint traces of the original image remain visible. This aligns with the expected outcome of the forward process, where larger timesteps correspond to more noise and less of the original image content.

Images at different timesteps

t=250



t=500



t=750

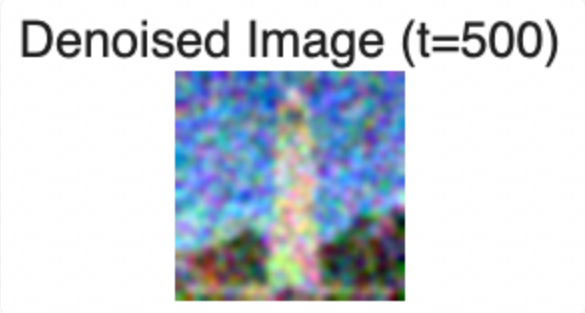
Part 1.2: Classical Denoising

In this section, I implemented a classical approach to image denoising using Gaussian blur filtering. I created a function called denoise_with_gaussian_blur that takes a noisy image as input and utilizes torchvision's gaussian_blur implementation to perform the denoising operation.

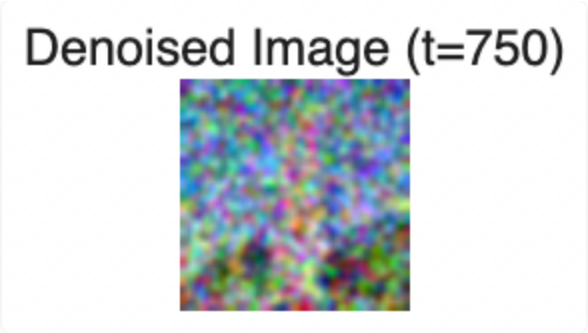
I applied this denoising function to the previously generated noisy images at timesteps 250, 500, and 750. The results showcase the limitations of classical denoising methods: while the Gaussian blur was somewhat effective for the lightly noised image (t=250), it struggled with the more heavily noised images. For t=750, the denoising was largely ineffective, as it was unable to recover meaningful image content from the heavily corrupted input.

Denoised Images at Different Timesteps

t=250



t=500



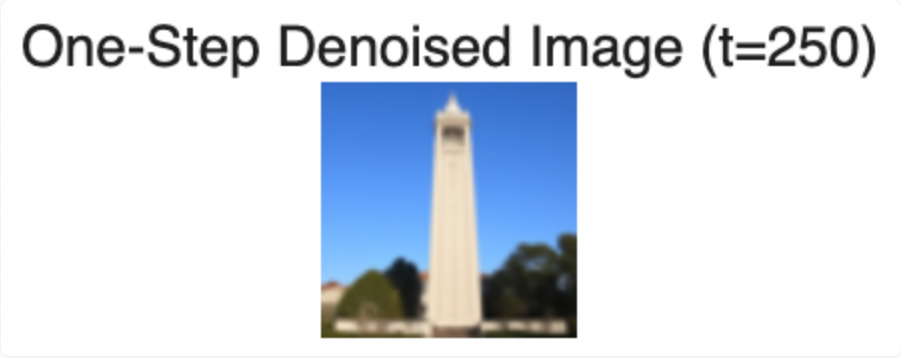
t=750

Part 1.3: One-Step Denoising

For each of the specified timesteps, the code calculates a scaling factor based on the alphas_cumprod array. A noisy version of the original image is then generated through a forward diffusion process. This noise estimate is then scaled appropriately and subtracted from the noisy image to recover the denoised image.

This visualization demonstrates the effectiveness of the denoising algorithm as the image becomes progressively clearer at different levels of noise. The results are shown for timesteps 250, 500, and 750, providing a clear depiction of how the one-step process is able to restore the original images.

One-Step Denoised Images at Different Timesteps



t=250

t=500

t=750

Part 1.4: Iterative Denoising

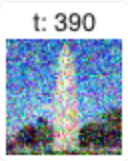
In this implementation, we worked on optimizing the process by skipping intermediate timesteps instead of performing a full sequence of denoising steps. The goal was to iteratively estimate and denoise an image through strided timesteps, reducing noise progressively while maintaining computational efficiency. We started by constructing a list of strided timesteps, moving from the noisiest state to a clean image by using specific intervals (e.g., step sizes of 30). At each step, we applied the iterative denoising formula.

The results are displayed at specific intervals for better visualization, and comparisons are drawn with a one-step denoising approach to highlight improvements achieved by iterative refinement. This process showcases how strided iteration can effectively balance performance and efficiency when reconstructing clean images from noisy inputs.

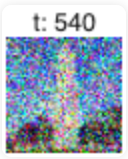
Iteratively Denoised Camapanile from t=90 to t=690

t=90

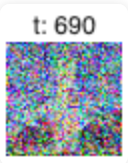
t=240



t=390



t=540



t=690

Other Methods

Original

Iteratively Denoised Campanile

One Step Denoised Campanile

Gaussian-Blurred Campanile

Part 1.5: Diffusion Model Sampling

This task involves generating images from random noise using a diffusion model. The process begins with initializing pure noise using torch.randn and then applies the iterative_denoise function, which gradually reduces noise to generate coherent images. By setting i_start = 0, the model starts the denoising process from the noisiest state and iterates until a clean, realistic image is produced.

The code executes this for five samples, creating a batch of images. The denoised outputs are displayed to visualize the results, showing the model's capability to create reasonable image outputs from random noise, though their quality is subject to further refinement. This showcases the power of diffusion models in generative tasks.

Randomly Generated Samples

Sample 1

Sample 2

Sample 3

Sample 4

Sample 5

Part 1.6: Classifier-Free Guidance (CFG)

This task aims to improve the quality of generated images using Classifier-Free Guidance (CFG). In this approach, the noise estimates are adjusted by combining a conditional noise estimate (generated using a given text prompt) and an unconditional noise estimate (generated with a null prompt). The two estimates are blended, with the difference scaled by a guidance factor. This scaling helps steer the generation process toward the desired prompt, producing images that are more coherent and aligned with the intended content.

The function iterative_denoise_cfg builds on the earlier denoising process but adds an extra step to compute both conditional and unconditional noise estimates at each iteration. The model is run twice per timestep — once with the prompt and once with a null prompt—to obtain these estimates. Using the CFG formula, the combined noise estimate is used to iteratively refine the image. This process results in higher-quality images compared to the previous approach. This demonstrates the effectiveness of CFG in guiding diffusion models for better image generation.

Randomly Generated Samples with CFG

Sample 1 with CFG

Sample 2 with CFG

Sample 3 with CFG

Sample 4 with CFG

Sample 5 with CFG

Part 1.7: Image-To-Image Translation

This task focuses on image-to-image translation using diffusion models. By adding noise to an existing image and then denoising it while conditioning on a text prompt, the model effectively edits the image. The extent of the edit is determined by the amount of noise added: higher noise levels result in more creative edits, while lower levels retain more of the original image's structure. This follows the SDEdit algorithm, which leverages the denoising process to force noisy images back onto the natural image manifold.

The implementation involves first applying noise to the test image at different levels, represented by the starting index values [1, 3, 5, 7, 10, 20]. For each noise level, the iterative_denoise_cfg function is used with both conditional and unconditional prompts. The conditional prompt guides the model during denoising, while the CFG scale adjusts the influence of the conditional guidance. The results are stored and visualized, showing a

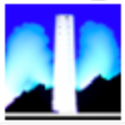
progression from heavily edited images (higher noise levels) to ones closer to the original image (lower noise levels).

Samples Generated with the SDEdit Algorithm

SD Edit with i_start=1

SD Edit with i_start=3

SD Edit with i_start=5



SD Edit with i_start=7

SD Edit with i_start=10

SD Edit with i_start=20

SD Edit with i_start=1

SD Edit with i_start=3

SD Edit with i_start=5

SD Edit with i_start=7

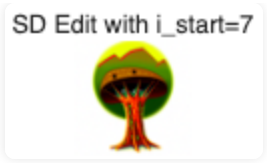
SD Edit with i_start=10

SD Edit with i_start=20

SD Edit with i_start=1

SD Edit with i_start=3

SD Edit with i_start=5

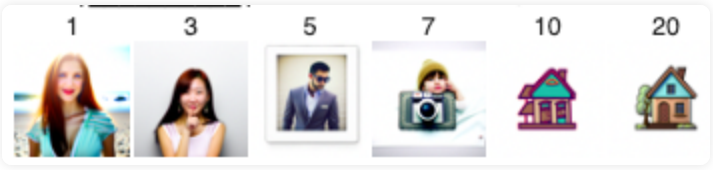


Part 1.7.1 Editing Hand-Drawn and Web Images

This task extends the image-to-image translation process to both hand-drawn images and images downloaded from the web, aiming to transform these into realistic outputs guided by the text prompt "a high quality photo."

For web images, the task involves downloading an image, preprocessing it into the required format, adding varying levels of noise, and applying the iterative denoising process to generate progressively realistic edits. For hand-drawn images, the user is prompted to create their own sketches, preprocess them similarly, and apply the same steps.

Web Image Denoised



Original

First Drawing Denoised

Original

Second Drawing Denoised

Original

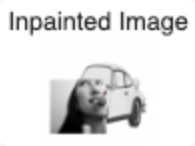
Part 1.7.2 Inpainting

This task focuses on inpainting, where specific regions of an image are reconstructed or filled in using a diffusion model. The process uses a binary mask to specify which parts of the image to edit. In regions where the mask equals 1, new content is generated based on the prompt, while areas where the mask equals 0 are preserved to maintain the original image.

At each timestep, after generating a denoised image, the function enforces that the non-masked regions remain consistent with the original image. This is achieved by blending the current denoised output with the original image using the mask. The function also utilizes Classifier-Free Guidance (CFG) to steer the generation process.

Campanile Inpainted

Car Inpainted



Tree Inpainted

Part 1.7.3 Text-Conditional Image-to-image Translation

This task expands on the previous image-to-image translation process by introducing a text condition to guide the edits. By conditioning the denoising process on a prompt such as "a rocket ship," the generated images are influenced not only by the original content but also by the semantics of the text. This results in progressively altered images that incorporate features aligned with the given text prompt.

At lower noise levels, the output images retain more of the original structure, while at higher noise levels, the influence of the text prompt becomes more pronounced, transforming the image into something that visually aligns with the idea of the prompt. This process is repeated for other test images, showcasing the flexibility of the text-conditioned image-to-image translation approach.

Rocket -> Campanile Text-Conditional Images

Edited Image - Noise Level 1

Edited Image - Noise Level 3

Edited Image - Noise Level 5

Edited Image - Noise Level 7

Edited Image - Noise Level 10

Edited Image - Noise Level 20

Rocket -> Tree Text-Conditional Images

Edited Image - Noise Level 1

Edited Image - Noise Level 3

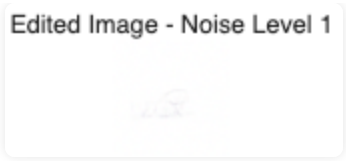
Edited Image - Noise Level 5

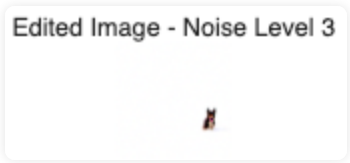
Edited Image - Noise Level 7

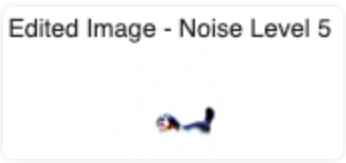
Edited Image - Noise Level 10

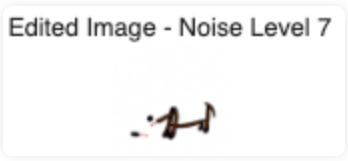
Edited Image - Noise Level 20

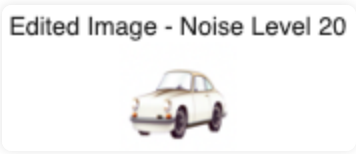
Dog -> Car Text-Conditional Images











Part 1.8 Visual Anagrams

This task involves creating visual anagrams, an application of diffusion models where the appearance of an image changes based on its orientation. The goal is to generate an image that, when viewed normally, represents one concept (e.g., "an oil painting of people around a campfire") and, when flipped, represents another concept (e.g., "an oil painting of an old man"). This is achieved by leveraging two different text prompts during the denoising process.

The visual_anagrams function implements this process by iterating through the timesteps,

combining the flipped and normal noise estimates, and performing denoising with the averaged noise. The exact algorithm for this involves two steps, first denoising the image normally using the first prompt to obtain a noise estimate, and second, flipping the image, applying the second prompt, flipping the resulting noise estimate back, and averaging the two noise estimates. The result is an image that seamlessly transitions between two visual concepts depending on its orientation.

Visual Anagrams

An oil painting of an old man & An oil painting of people around a campfire

An photo of the Amalfi Coast & A man wearing a hat

A photo of a dog & A photo of a hipster barista

Part 1.9 Hybrid Images

The goal of this task is to produce an image that appears as one concept from afar, such as "a skull," and transitions to another concept, like "a waterfall," when viewed up close. This effect is achieved by combining low-frequency information (captured via a Gaussian blur or low-pass filter) from one prompt with high-frequency details (captured via a high-pass filter) from another prompt.

A low-pass filter is applied to the noise estimate corresponding to the first prompt, while a high-pass filter extracts details from the second prompt. These filtered noise components are then combined into a composite noise map. The hybrid image is created by iteratively

denoising using the composite noise map, guided by the combined frequencies. The blending allows the image to convey distinct interpretations based on the viewer's perspective. Additional hybrid images with other prompt combinations further demonstrate this process.

Hybrid Images

Skull from afar, Waterfall up close

Amalfi Coast from afar, Dog up close

Pencil from afar, Rocket up close

Saatvik Billa CS180 Project 5B: Diffusion Models from Scratch!

Part 1.1: Implementing the UNet

The first step in this project was to implement and train a single-step U-Net as a denoiser. The objective was to map a noisy image to its clean counterpart by minimizing the difference between the denoised output and the target clean image. This was achieved by optimizing the L2 loss function, which calculates the mean squared error between the predicted and target images.

The U-Net architecture used in this implementation is designed with a symmetric encoder-decoder structure and skip connections. The encoder consists of convolutional and downsampling blocks that progressively reduce the spatial resolution while capturing feature hierarchies. At the bottleneck, the spatial dimensions are flattened to create a compact representation. The decoder then upscales these features back to the original resolution using upsampling blocks, and skip connections ensure that spatial details from the encoder are preserved.

The implementation includes a Python class, UnconditionalUNet, which defines the architecture and forward pass. The encoder uses two downsampling blocks and convolutional layers to encode features. The bottleneck processes the encoded features into a flattened representation, while the decoder reconstructs the output by upsampling and concatenating features via skip connections. The forward pass computes intermediate outputs at each stage of the encoder and merges these with the corresponding outputs from the decoder to reconstruct the final clean image.

Part 1.2: Using the UNet to Train a Denoiser

Part 1.2.1: Training

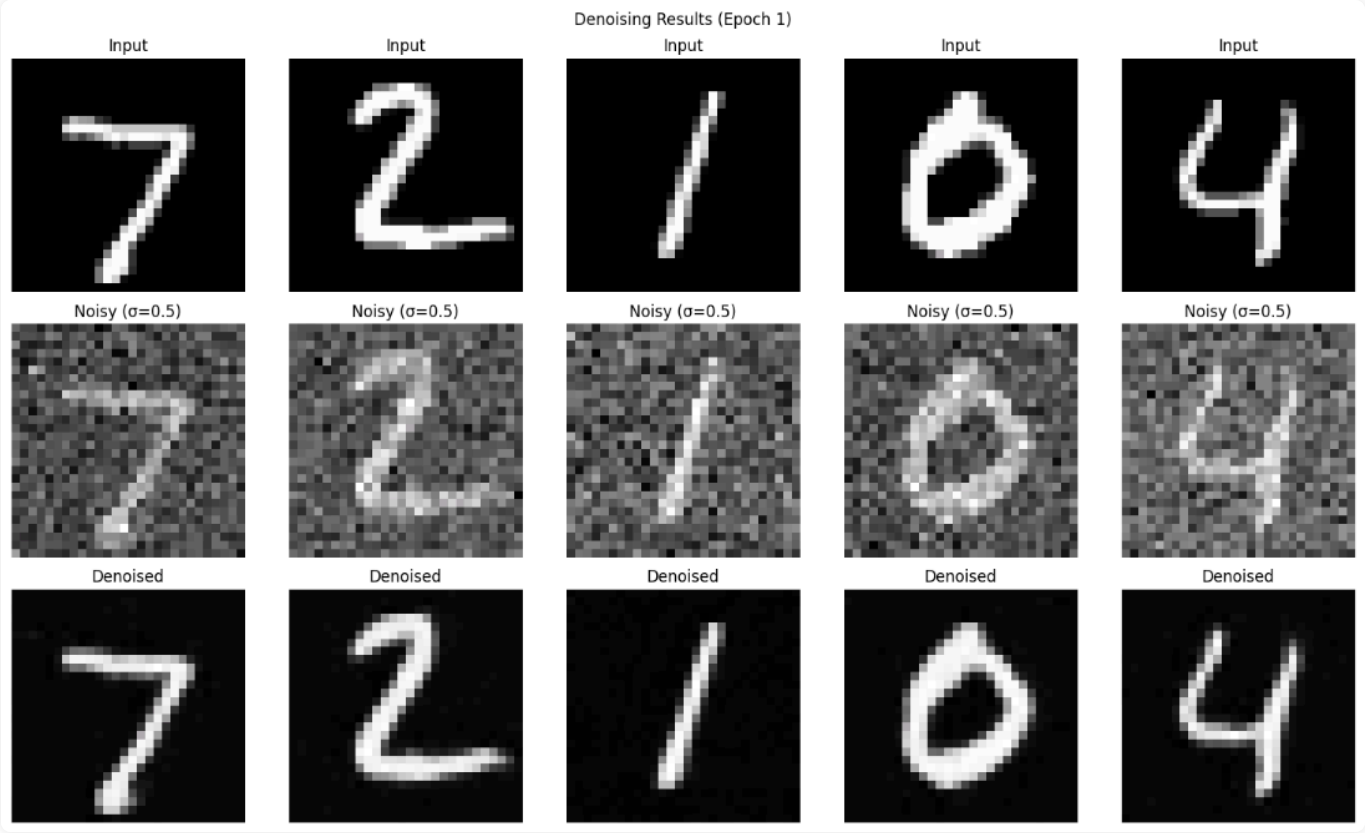
Noisy images were generated by adding Gaussian noise with a standard deviation of 0.5 to clean MNIST images. The objective was to minimize the mean squared error (MSE) between the U-Net's output and the clean images.

When training, the dataset was used with a batch size of 256. To ensure generalization, noise was added to image batches dynamically during each training epoch, preventing the model from overfitting to fixed noisy inputs. The training process used the Adam optimizer with a learning rate of 0.0001 over five epochs. For each batch, noisy images were fed into the U-Net, the loss was computed using the MSE criterion, and the weights were updated through backpropagation.

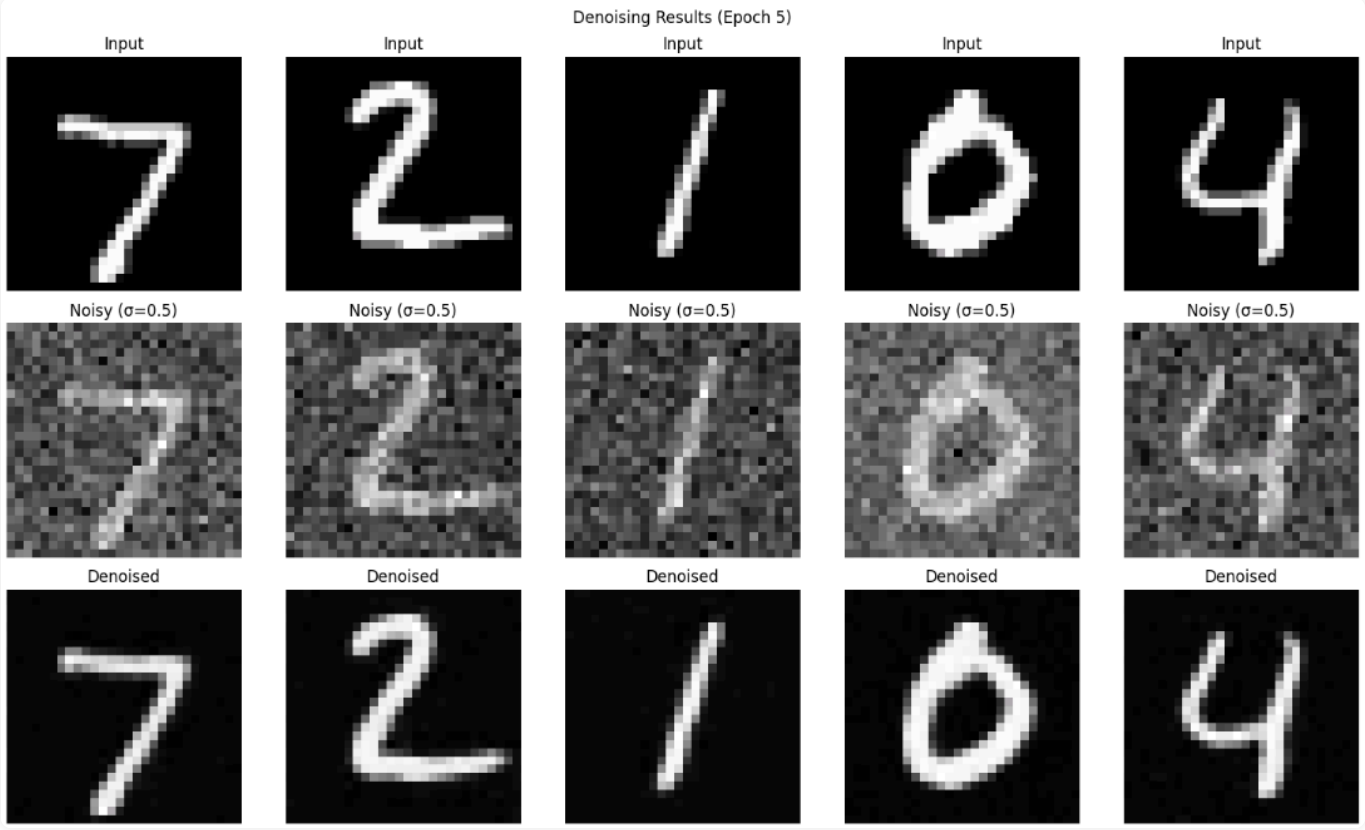
As training progressed, the loss curve demonstrated a clear downward trend, reflecting the model's improving denoising capability. The setup highlights the U-Net's ability to handle

noisy inputs and reconstruct clean images effectively, highlighting its potential as a denoising model.

Denoising Results after 1 Epoch



Denoising Results after 5 Epochs

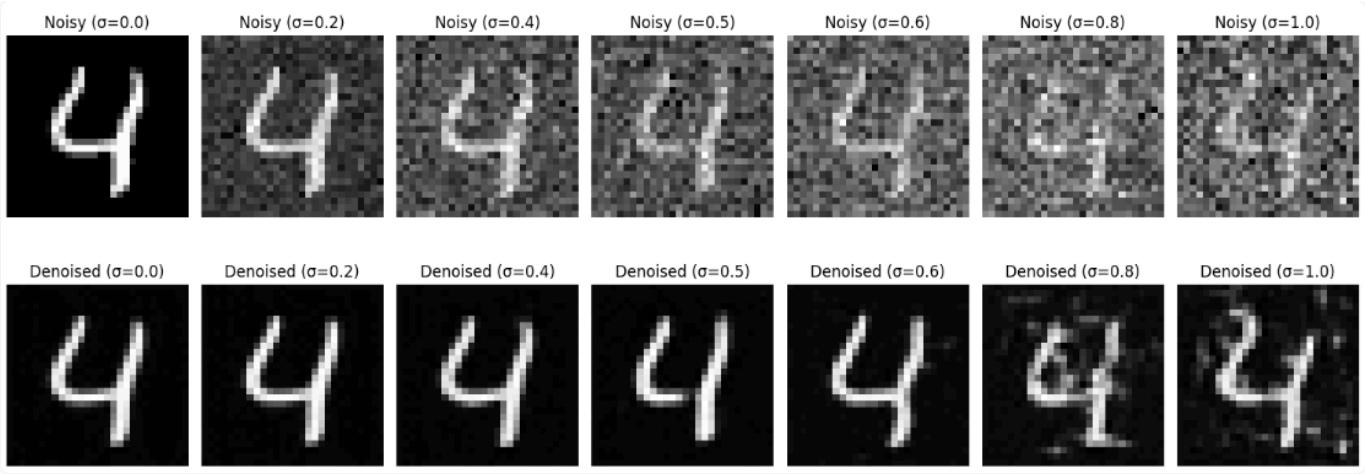


1.2.2 Out-of-Distribution Testing

The denoiser's performance is evaluated on test set images with different levels of noise (sigma values ranging from 0 to 1.0). The implementation visualizes both the noisy images and their corresponding denoised outputs, comparing the effect of noise intensity on the model's ability to restore clean digits.

For each noisy image, the model predicts the denoised version, which is then displayed alongside the noisy image for visual comparison. This process demonstrates its effectiveness at low to moderate noise (sigma <= 0.5) and showing its limitations as the noise becomes more intense (sigma > 0.5). The results indicate that while the model performs well under conditions similar to its training, its ability to generalize to higher noise levels diminishes, as seen in the more distorted denoised outputs at sigma = 0.8 and sigma = 1.0.

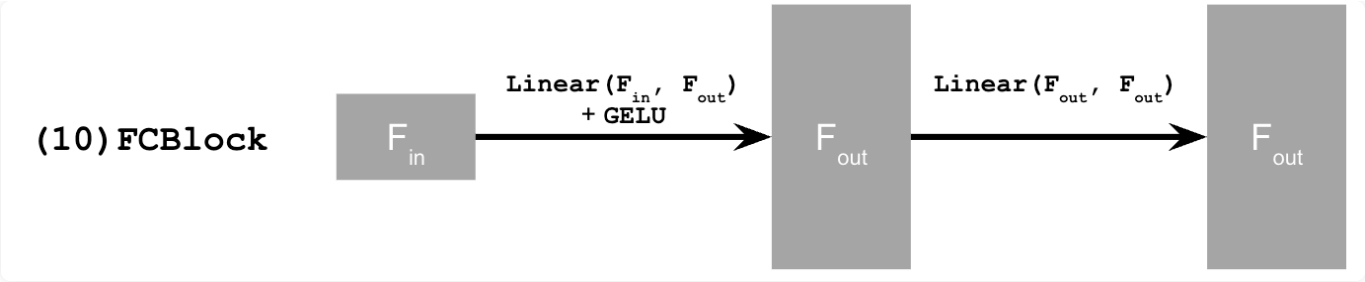
Denoised Results for Various Sigmas



Part 2.1: Adding Time Conditioning to UNet

To integrate time-conditioning into the U-Net architecture, a scalar timestep (t) is injected into the network using fully connected blocks (FCBlocks). These blocks transform the scalar timestep into a feature representation that can be incorporated into the U-Net's architecture. The time-conditioning signal is added at multiple stages of the network by concatenating it with the intermediate feature maps, enabling the model to leverage temporal information effectively during processing.

The U-Net's architecture consists of standard convolutional blocks for encoding and decoding, along with time-conditioning via the FCBlocks. Each FCBlock is implemented using two linear layers, with GELU activations in between to ensure smooth and non-linear transformations of the timestep. These FCBlocks are strategically placed in the bottleneck and decoder stages of the U-Net. The processed time information is injected at these stages by concatenating it with the existing feature maps, ensuring that the timestep influences the spatial and feature-level transformations throughout the network.

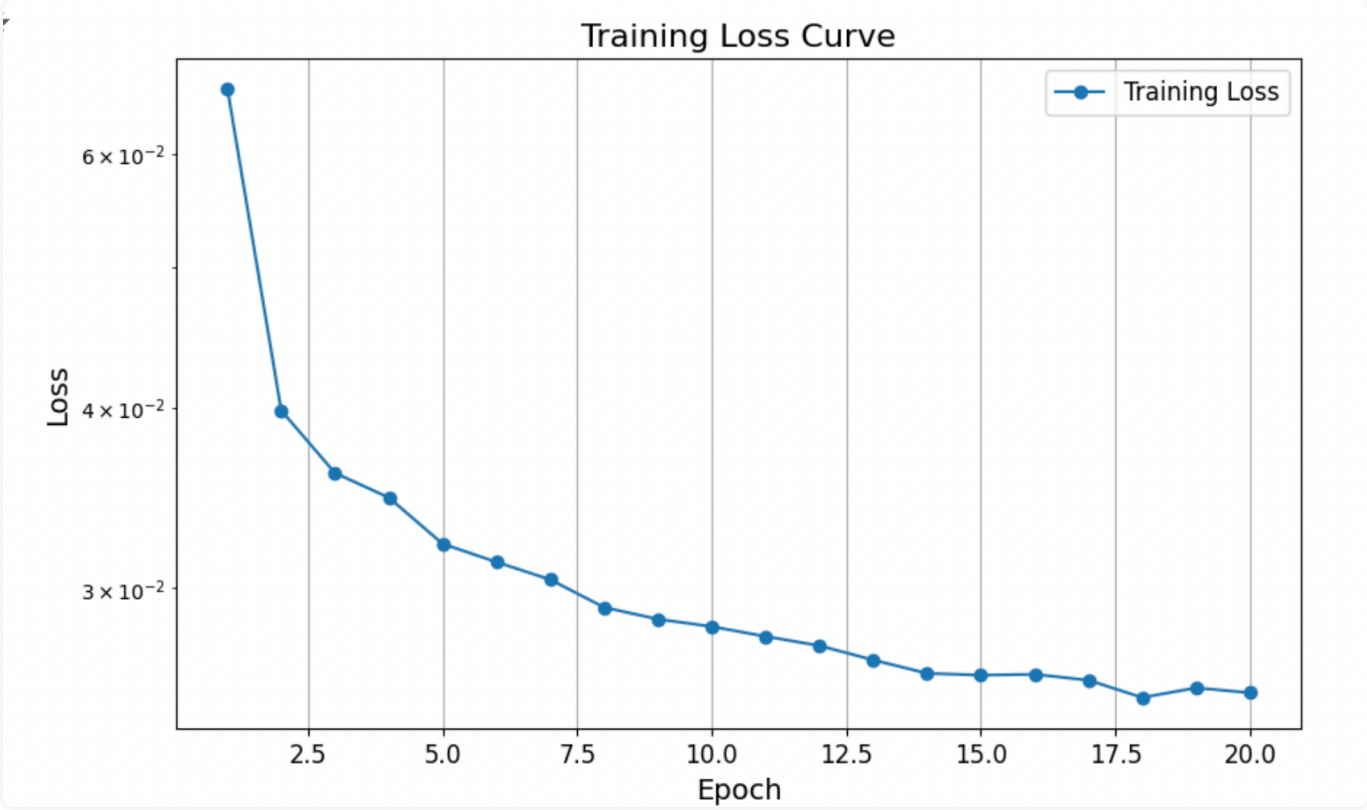


Newly Introduced FCBlock Architecture (taken from spec)

Part 2.2: Training the UNet

The objective of this task was to train a time-conditioned U-Net to predict noise in noisy images at a specific timestep using the MNIST dataset. This involved iteratively sampling random images, adding noise to them based on the given timestep, and training the model to recover the noise component. The U-Net was designed to incorporate the timestep as a normalized conditioning signal embedded into the model. The training process utilized a batch size of 128 and spanned 20 epochs. The Adam optimizer was employed with an initial learning rate of 0.001, while an exponential learning rate scheduler was applied to decay the learning rate smoothly over the course of training.

During training, noisy image batches were passed through the model to compute the loss, which was then used for backpropagation to update the model weights. The process was repeated for all batches in each epoch, with the training loss logged for monitoring. Model checkpoints were saved at key intervals, specifically after the 5th and 20th epochs, to ensure progress was preserved. Finally, the training loss curve was plotted on a logarithmic scale to visualize the convergence of the model over time.

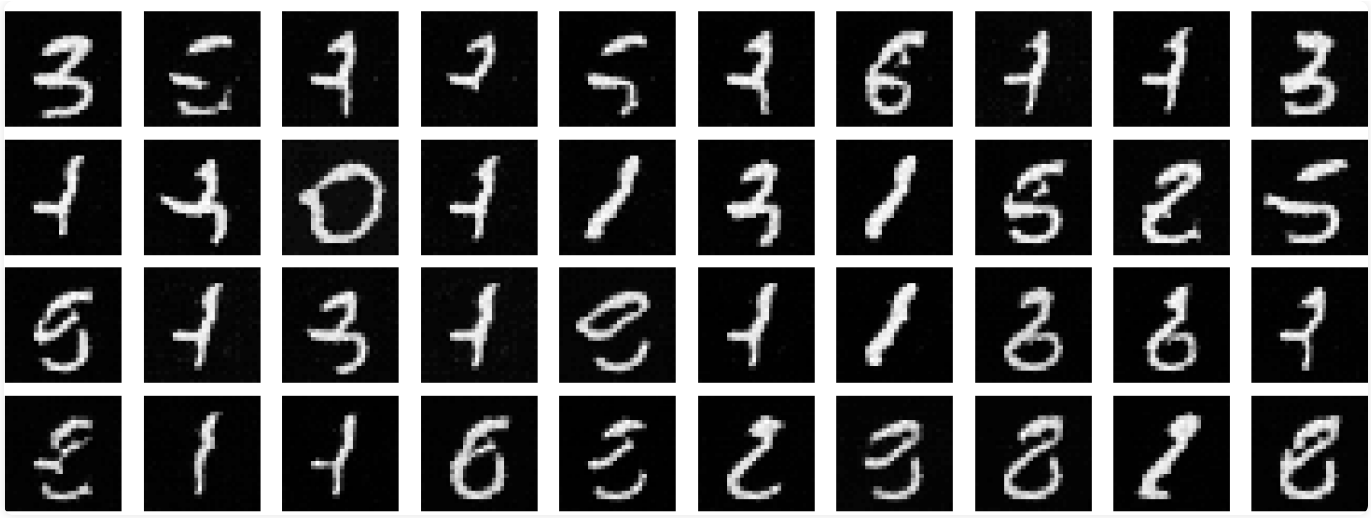


TimeConditionalUNet Training Loss Over 20 Epochs

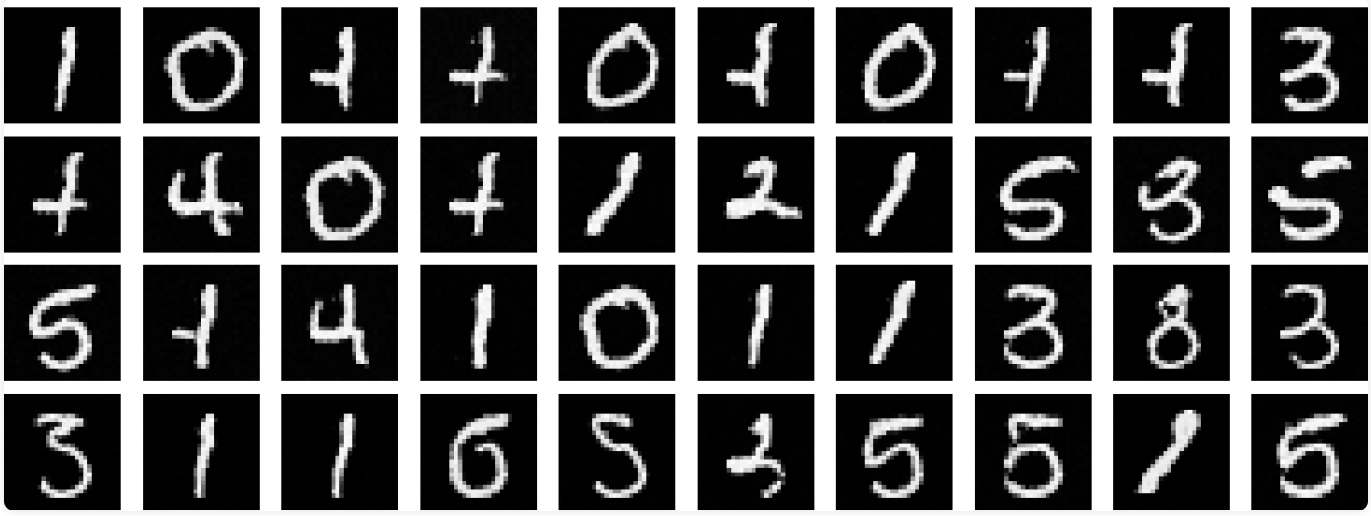
Part 2.3: Sampling from the UNet

The sampling process for generating images using the time-conditioned U-Net was based on the reverse diffusion process. This approach starts with pure Gaussian noise and iteratively refines it by denoising through the model. The key steps include precomputing the scheduling parameters (beta, alpha, and alpha bar), initializing a noisy input, and iteratively denoising it over the predefined timesteps. The sampling algorithm refines the noisy image at each step by predicting and subtracting noise using the U-Net model and combines it with precomputed coefficients to reconstruct the intermediate image.

Images sampled from early epochs (e.g., Epoch 5) appear less defined, while those from later epochs (e.g., Epoch 20) exhibit significantly improved clarity and structure, reflecting the model's learning progress. This sampling approach effectively leverages the trained U-Net to generate high-quality image samples from Gaussian noise.



Samples after 5 Epochs

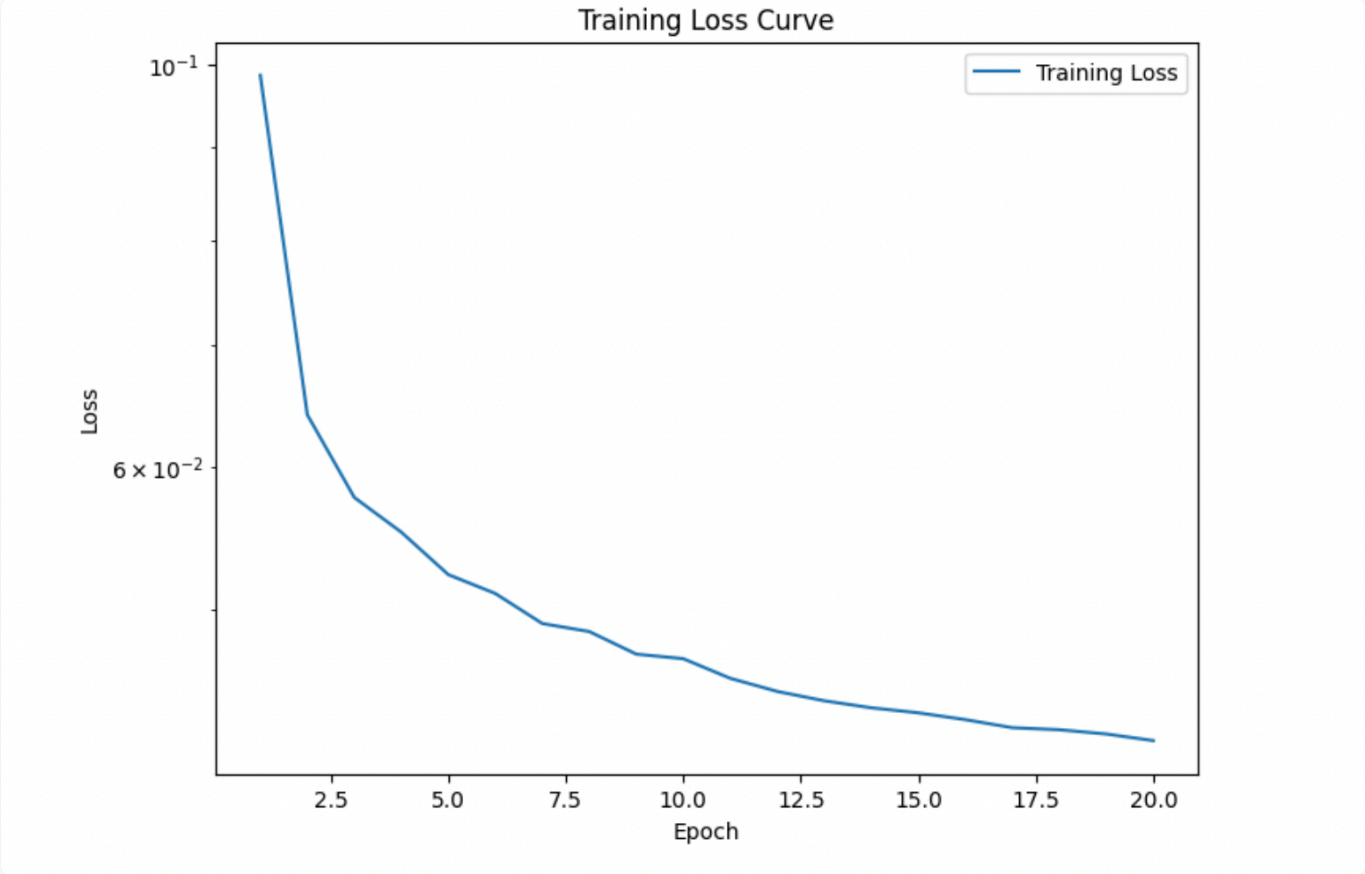


Samples after 20 Epochs

Part 2.4: Adding Class-Conditioning to UNet

To improve the quality and control of image generation, class-conditioning was added to the U-Net model. This involved conditioning the model on the digit class (0-9) in addition to the timestep. Two additional fully connected blocks (FCBlocks) were introduced to encode the class-conditioning vector as part of the architecture. The class-conditioning vector was represented as a one-hot encoded vector, ensuring it effectively captured class-specific information. To maintain flexibility, dropout was implemented such that 10% of the time, the class-conditioning vector was set to zero, allowing the model to generate unconditioned outputs.

The training loop followed the same structure, where noisy images were generated, and the model was trained to predict the noise component. Loss calculations and model updates were performed as before, and checkpoints were saved after the 5th and 20th epochs. The addition of class-conditioning enhanced the model's ability to generate more controlled outputs based on the specified class. As with the time-conditioned U-Net, the training loss was tracked and visualized using a logarithmic scale to monitor convergence over the epochs.

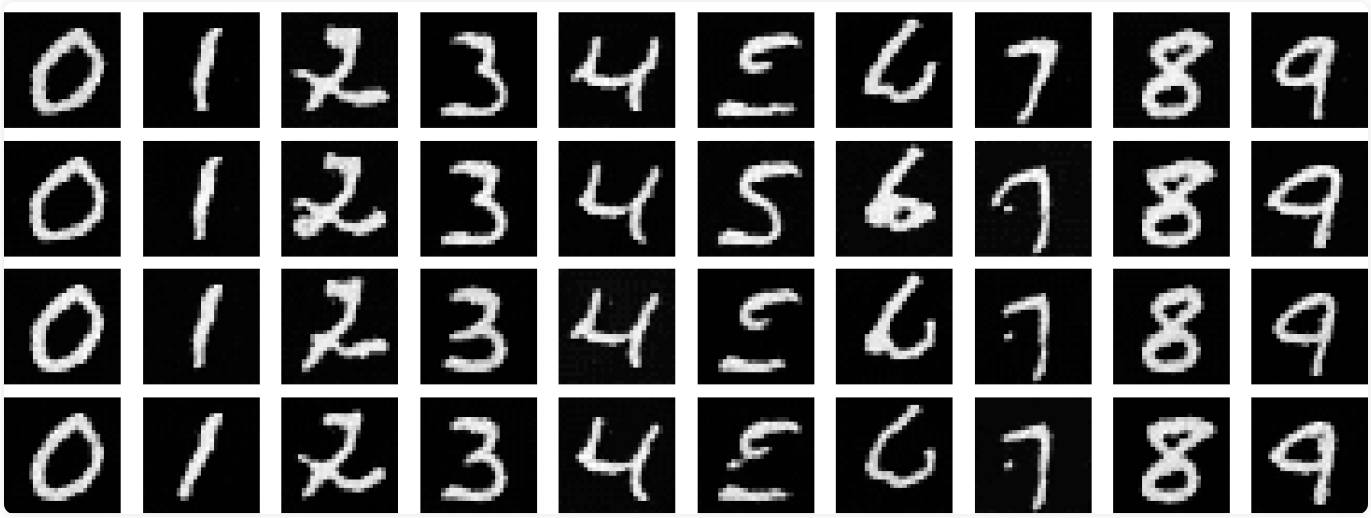


ClassConditionalUNet Training Loss Over 20 Epochs

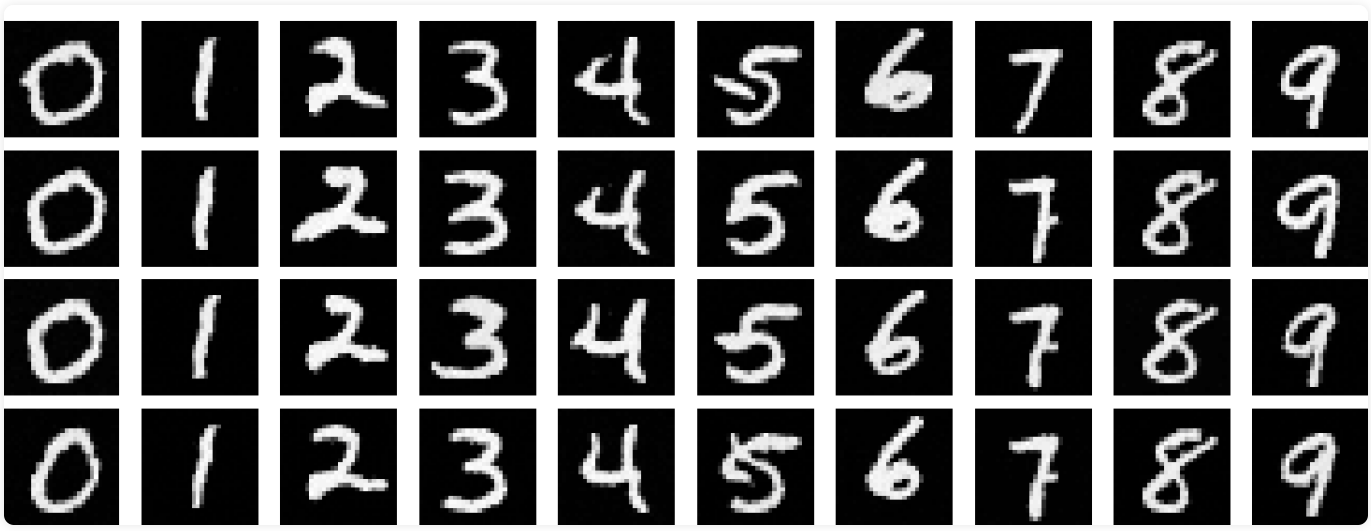
Part 2.5: Sampling from the Class-Conditioned UNet

The results, visualized across epochs, show the effectiveness of this approach. At Epoch 1, the generated digits are poorly formed and ambiguous. By Epoch 5, the outputs are significantly more refined, with digits clearly representing their respective classes. This

method effectively leverages class-conditioning and classifier-free guidance to produce high-quality, class-specific samples.



Samples after 5 Epochs



Samples after 20 Epochs

Takeaways

This project was such a cool dive into the power of diffusion models. I got to see how they can take pure noise and turn it into stunning, realistic images through various methods, such

as iterative denoising, CFG, and the coolest was definitely UNets. Can't wait to put this on my portfolio!