Part 1: Noise Scheduler

Description

The Noise Scheduler controls the forward diffusion process by defining how noise is progressively added to clean images. I implemented a **linear beta schedule** where β_t increases linearly from `beta_start = 1e-4` to `beta_end = 0.02` over `T = 1000` timesteps. From the betas, we derive α_t = 1 − β_t and the cumulative products $\bar{\alpha}_t = \prod \alpha_s$, along with their square roots, which are used to sample noisy images at any arbitrary timestep in closed form: $x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{(1 - \bar{\alpha}_t)} \cdot \varepsilon$.

Evaluation

The forward diffusion process visualization confirms correct behavior: at t = 0 the image is clean, structure degrades gradually through t = 250 and t = 500, and by t = 999 the image is indistinguishable from pure noise.

Part 2: Sinusoidal Position Embedding

Description

The time embedding module encodes the discrete integer timestep t into a continuous vector of dimension 128 using **sinusoidal positional encodings**, identical in form to those used in Transformers. Even dimensions use sin and odd dimensions use cos, with exponentially spaced frequencies. This embedding is then projected through a small MLP (Linear → SiLU → Linear) to produce the final time conditioning vector.

Part 3: U-Net Architecture

Description

The backbone of the diffusion model is a **simplified U-Net** with the following structure:

- **Initial convolution:** 1 → 64 channels
- **Encoder:** Two downsampling stages with ConvBlocks (64 → 64, 64 → 128), each followed by 2× max-pooling. Each ConvBlock consists of two Conv2d layers with GroupNorm (8 groups) and SiLU activation, plus an additive time-embedding injection.
- **Middle block:** A ConvBlock at the bottleneck (128 → 256 channels)
- **Decoder:** Two upsampling stages with bilinear interpolation, skip connections concatenated from the encoder, and ConvBlocks (256+128 → 128, 128+64 → 64)
- **Output convolution:** 64 → 1 channel

The total model has **1,011,969 trainable parameters**.

Part 4: Training

Description

The model is trained by minimizing the **simplified DDPM objective** (mean squared error between the true noise ε and the predicted noise ε_θ(x_t, t)). At each training step, random timesteps are sampled uniformly from [0, T), noise is added to the clean images via the scheduler, and the model predicts the noise.

- **Optimizer:** AdamW with learning rate 2e-4
- **Batch size:** 128
- **Epochs:** 10

Parameter Justification: AdamW at 2e-4 is the standard optimizer/learning rate combination for DDPM training, providing stable convergence with weight decay regularization. A batch size of 128 balances GPU memory usage and gradient noise. Training for 10 epochs is sufficient for MNIST to achieve recognizable digit generation given the relative simplicity of the dataset.

Part 5: Sampling (Image Generation)

Description

Image generation follows the DDPM reverse process: starting from pure Gaussian noise x_T, the model iteratively denoises through all 1000 timesteps. At each step, the model predicts the noise component, which is subtracted (scaled by the appropriate coefficients), and stochastic noise $\sigma_t \cdot z$ is added for all steps except the final one (t = 0).

Part 6: Visualize the Reverse Diffusion Process

Description

To better understand the generation process, intermediate states are saved at timesteps t = 999, 900, 750, 500, 250, 100, 50, and 0 during sampling.

Reflection

This lab provided a hands-on understanding of how diffusion models work from the ground up. A few key takeaways:

The elegance of the DDPM framework is striking — the forward process has a clean closed-form solution for any timestep, and the training objective reduces to a simple noise prediction MSE loss. Despite this simplicity, the model can generate realistic-looking digits after just 10 epochs of training.

Overall, building a DDPM from scratch demystified the core mechanism behind state-of-the-art image generators and provided a solid foundation for understanding more advanced diffusion-based methods.