# CVPDL HW3 Report

R14921114 紀敦翊

## 1 Model Description

I implement a Denoising Diffusion Probabilistic Model (DDPM) tailored to MNIST (28×28) using a time-conditioned U-Net. The model predicts the additive noise $\epsilon$ at each diffusion step (the "$\epsilon$-prediction" objective). A linear $\beta_t$ schedule is used to define the forward noising and the reverse denoising processes.

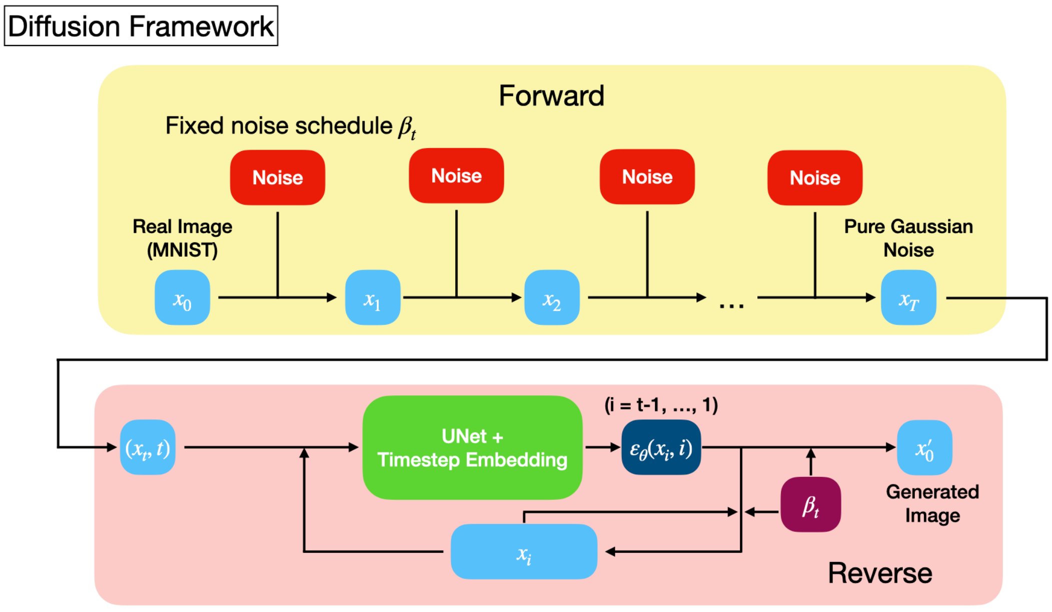


Figure 1: Diffusion Framwork

### 1.1 Forward Diffusion

I use a linear noise schedule with $T = 800$ timesteps. The scheduler first creates a sequence of betas $\beta \in [10^{-4}, 0.02]$, then defines $\alpha_t = 1 - \beta_t$ and their cumulative product $\bar{\alpha}_t = \prod_{s=0}^{t} \alpha_s$. Instead of simulating every step $x_{t-1} \to x_t$, I use the closed-form distribution

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

implemented in `forward_diffusion`. For each training image I randomly sample a timestep $t$ and generate a noisy sample $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{(1 - \bar{\alpha}_t)}\varepsilon$, where $\varepsilon \sim \mathcal{N}(0, 1)$. The scheduler class stores $\beta_t, \alpha_t, \bar{\alpha}_t$, and their square roots so they can be reused efficiently during both training and sampling.

### 1.2 Reverse Denoising Model

The reverse process is parameterized by a 2D UNet with timestep conditioning, shown in the UNet diagram. At each denoising step the network receives $(x_t, t)$ and predicts the added noise

$\varepsilon_\theta(x_t, t)$ with the same shape as the input image. The predicted noise is then plugged into the DDPM mean formula for

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(\mu_\theta(x_t, t), \sigma_t^2 \mathbf{I}),$$

where $\mu_\theta(x_t, t)$ is computed exactly as in the original DDPM paper and $\sigma_t^2$ uses the closed-form posterior variance. During sampling I start from $x_T \sim \mathcal{N}(0, 1)$ and run this reverse process iteratively from $t = T - 1$ down to 0 to obtain the generated image $x'_0$.

## 1.3 UNet Backbone

The UNet operates on 3-channel 28×28 RGB images. The encoder–decoder architecture uses a base channel width of 128 and channel multipliers (1,2,4). As illustrated:

- The downsampling path contains two DownBlocks. Each block has two residual blocks followed by a strided 3×3 convolution for spatial downsampling. Intermediate feature maps at resolutions 28×28 and 14×14 are stored as skip1 and skip2.
- The bottleneck ("Middle") consists of two residual blocks that increase channels from 256 to 512 and keep the resolution at 7×7.
- The upsampling path mirrors the encoder. Each UpBlock first upsamples by a factor of 2 (nearest-neighbor interpolation plus a 3×3 convolution), concatenates the corresponding skip feature, and passes the result through two residual blocks to fuse encoder and decoder information.
- The final head applies GroupNorm, SiLU, and a 3×3 convolution to map the 128-channel feature map back to 3 channels.

Every residual block uses GroupNorm with 8 groups, SiLU activations, and an extra linear layer applied to a sinusoidal timestep embedding. The timestep is first encoded by a standard sinusoidal positional embedding, then passed through a small MLP; the resulting vector is added to the intermediate feature maps in each residual block as a learned, timestep-dependent bias. This makes the UNet explicitly aware of the current diffusion step.

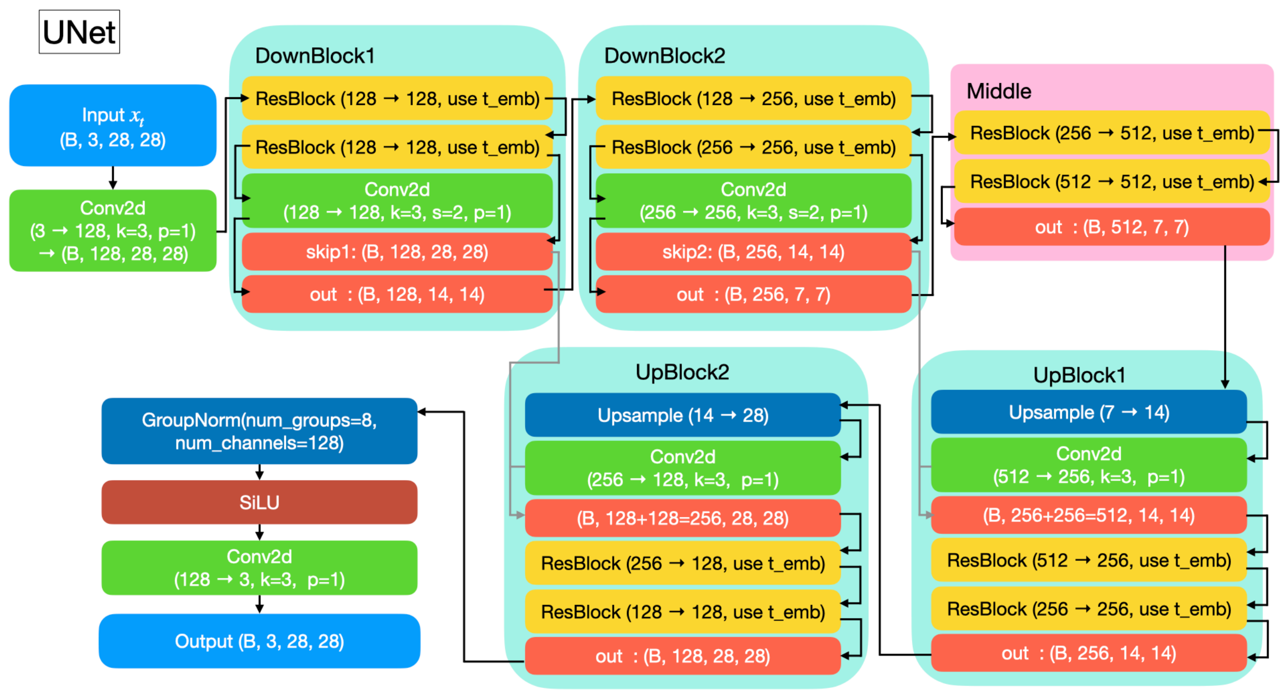

UNet
Input $x_t$ (B, 3, 28, 28)
Conv2d (3 → 128, k=3, p=1) → (B, 128, 28, 28)
DownBlock1
ResBlock (128 → 128, use t_emb)
ResBlock (128 → 128, use t_emb)
Conv2d (128 → 128, k=3, s=2, p=1)
skip1: (B, 128, 28, 28)
out : (B, 128, 14, 14)
DownBlock2
ResBlock (128 → 256, use t_emb)
ResBlock (256 → 256, use t_emb)
Conv2d (256 → 256, k=3, s=2, p=1)
skip2: (B, 256, 14, 14)
out : (B, 256, 7, 7)
Middle
ResBlock (256 → 512, use t_emb)
ResBlock (512 → 512, use t_emb)
out : (B, 512, 7, 7)
UpBlock1
Upsample (7 → 14)
Conv2d (512 → 256, k=3, p=1)
(B, 256+256=512, 14, 14)
ResBlock (512 → 256, use t_emb)
ResBlock (256 → 256, use t_emb)
out : (B, 256, 14, 14)
UpBlock2
Upsample (14 → 28)
Conv2d (256 → 128, k=3, p=1)
(B, 128+128=256, 28, 28)
ResBlock (256 → 128, use t_emb)
ResBlock (128 → 128, use t_emb)
out : (B, 128, 28, 28)
GroupNorm(num_groups=8, num_channels=128)
SiLU
Conv2d (128 → 3, k=3, p=1)
Output (B, 3, 28, 28)

Figure 2: UNet Structure

# 2 Implementation Details

## 2.1 Dataset and Preprocessing

- Dataset: MNIST training set, resized to 28×28 and loaded via `torchvision.datasets.MNIST`.
- Transforms: Resize(28,28), ToTensor(), and Normalize((0.5,), (0.5,)). This normalizes the gray-scale values into $[-1, 1]$.
- Since MNIST is single-channel, I replicate the channel dimension to obtain 3-channel tensors before feeding them into the UNet, both during training and sampling.

## 2.2 Noise Scheduler

- Total timesteps: `num_timesteps=800` (used consistently in training and sampling).
- Linear $\beta_t$ schedule from `beta_start=1e-4` to `beta_end=0.02`.
- Precomputes $\beta, \alpha, \bar{\alpha}, \sqrt{\bar{\alpha}}$, and $\sqrt{1-\bar{\alpha}}$ on the target device.
- Provides a helper function `add_noise(x0, noise, t)` used internally and a standalone `forward_diffusion` function implementing the closed-form $x_t$ sampling.

## 2.3 Training Objective and Procedure

- Objective: For each batch, I sample timesteps $t \sim \mathcal{U}\{0, ..., T-1\}$, generate $x_t$ and the corresponding noise $\varepsilon$ with `forward_diffusion`, and minimize the mean-squared error

$$\mathcal{L} = \mathbb{E}_{x_0,t,\varepsilon}[\| \varepsilon - \varepsilon_\theta(x_t, t) \|_2^2]$$

- Optimizer: Adam (`torch.optim.Adam`) with learning rate 1e-4, no weight decay, and no learning-rate scheduler.
- Hyperparameters: `epochs` = 150, `batch_size` = 256, `time_emb_dim` = 256, `base_ch` = 128, `channel_mults` = (1, 2, 4)
- Training loop: For each epoch, I iterate over the MNIST dataloader, compute the noisy images and target noise, run the UNet, compute the MSE loss, backpropagate, and update the parameters. The loop is wrapped with tqdm to log the current loss.
- Checkpointing: After training, the model weights are saved to `ddpm_mnist.pth` under the directory specified by `--save_model_path` (default `./img_gen/`).

## 2.4 Sampling and Image Generation

- For evaluation, I load the trained UNet using `load_model` and recreate a NoiseScheduler with the same number of timesteps and beta range.
- The `ddpm_sample` function starts from Gaussian noise $x_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively applies the DDPM reverse update:
  - Compute $\varepsilon_\theta(x_t, t)$ with the UNet.
  - Compute the mean $\mu_\theta(x_t, \varepsilon_\theta)$ using the closed-form DDPM formula.
  - Sample $x_{t-1}$ from $\mathcal{N}(\mu_t, \sigma_t^2 \mathbf{I})$ for $t > 0$; use the mean only at $t = 0$.
- After the loop, I map pixel values from $[-1, 1]$ back to $[0, 1]$, then save them as PNG files.

- Generation setup: by default I generate 10,000 images with a sampling batch size of 256. Images are stored in a user-specified output directory with 3 channels and resolution 28×28, matching the submission requirements.

## 3 Result Analysis

### 3.1 Quantitative Improvements

To study how the sampling length affects generation quality, I evaluate the FID under different numbers of diffusion timesteps with a fixed seed for training. For each timestep setting, I run the sampler three times with different random seeds, generate 10,000 images per run and compute the FID for each run.

| `num_timesteps` | run1 | run2 | run3 |
|---|---|---|---|
| 100 | 129.358 | 129.604 | 129.051 |
| 300 | 26.445 | 26.501 | 26.323 |
| 500 | 9.741 | 9.926 | 9.998 |
| 700 | 1.961 | 1.969 | 2.025 |
| 800 | 1.532 | 1.688 | 1.470 |
| 900 | 2.022 | 1.799 | 1.920 |
| 1000 | 5.892 | 6.020 | 5.849 |
| 1300 | 14.391 | 14.585 | 14.211 |

Table 1: FID of Different Timesteps

### 3.2 Quantitative Samples

In addition to the numerical FID scores, I visualize typical success and failure cases for the final setting. The figure shows examples of clearly recognizable digits that the model generates successfully, alongside failure samples whose shapes are noisy, ambiguous, or collapsed.

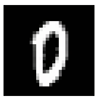
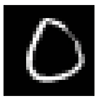
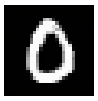
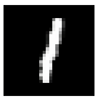
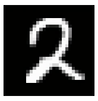
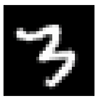
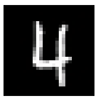
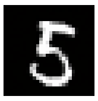
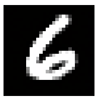
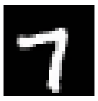
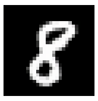
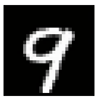

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

(i)

(j)

(k)

(l)

Figure 3: Success Samples

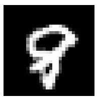
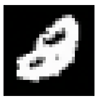
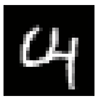
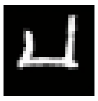
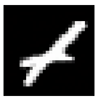
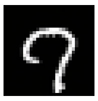

(a)

(b)

(c)

(d)

(e)

(f)

Figure 4: Failure Samples

### 3.3 Diffusion Process Visualization

To better understand how the model denoises images over time, I visualize the diffusion process starting from pure Gaussian noise. I randomly choose 8 different noise seeds and generate 8 corresponding samples. During sampling, I divide the total number of timesteps into 7 equal intervals and record the intermediate outputs at these checkpoints. The final visualization is an 8×8 grid (see figure): each row corresponds to one seed, the leftmost image is the initial noise, the middle images show intermediate states at increasing timesteps, and the rightmost image is the final generated digit at $t = 0$. From left to right, we can clearly observe how the structure of the digit gradually emerges from noise—some rows end with clean MNIST-like digits, while others still contain artifacts, reflecting both successful generations and occasional failures.

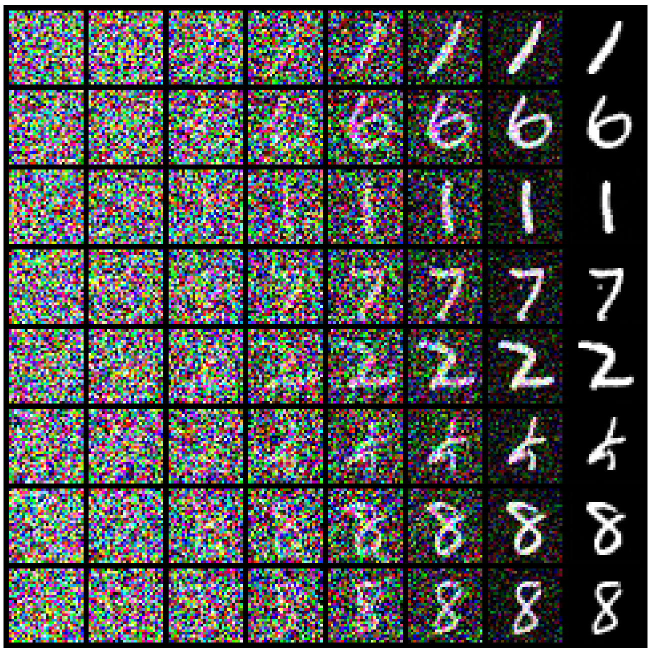


Figure 5: Diffusion Process Visualization

## 4 Short conclusion

In this homework I implemented a DDPM-style diffusion model with a timestep-conditioned UNet backbone for unconditional MNIST generation. Using a fixed linear noise schedule and an MSE loss on the predicted noise, the trained model is able to transform pure Gaussian noise into recognizable digit images. Quantitative experiments with different sampling timesteps, evaluated via FID over multiple runs, show that the choice of diffusion length has a clear impact on image quality, and that some configurations provide a better trade-off between fidelity and sampling cost than others.

The qualitative results further support these findings. Success and failure examples reveal that good samples closely match the shapes of real MNIST digits, while failure cases remain noisy or ambiguous. The diffusion process visualizations illustrate how structure gradually emerges from noise across timesteps, providing an intuitive view of how the reverse process works in practice. Overall, the experiments confirm that the implemented diffusion framework and UNet architecture can effectively model the MNIST data distribution, while leaving room for improvement through better noise schedules, model capacity, or sampling strategies.

## 5 Reference

1. The scratch of diffusion model

2. DDPM