Diffusion in Computer Vision

Oleh Prostakov[1]

Ukrainian Catholic University prostakov.pn@ucu.edu.ua

1 Introduction

The idea behind models covered in our second assignment is simple: take a clean image, gradually add Gaussian noise until it becomes pure static, and then train a neural network to reverse this process step by step.

At inference time, we start from random noise and iteratively denoise it to produce a new image.

In this project, we implemented, trained and compared different diffusion and flow models: DDPM and DDIM in the pixel-space; their adaptations for the latent space of VAE; their conditioned version; and, finally, Rectified Flow models in pixel- and latent-space.

2 DDPM and DDIM

2.1 Model Architecture

Our noise prediction network is a UNet [9] - the classic encoder-decoder with skip connections. We replicate the implementation of HuggingFace's `diffusers.UNet2DModel` and add a thin layer that handles MNIST's awkward 28×28 resolution (the UNet needs dimensions divisible by 2^L where L is the number of downsampling levels).

Architecture Overview The UNet processes a noisy image x_t together with a timestep embedding t and predicts the noise $\epsilon_\theta(x_t, t)$. Figure 1 shows the overall structure. Encoder (blue) downsamples through 4 resolution levels while decoder (red) upsamples back. Skip connections (dashed gray) preserve spatial detail. Self-attention is applied only at the 4×4 bottleneck (orange). The timestep embedding conditions all residual blocks.

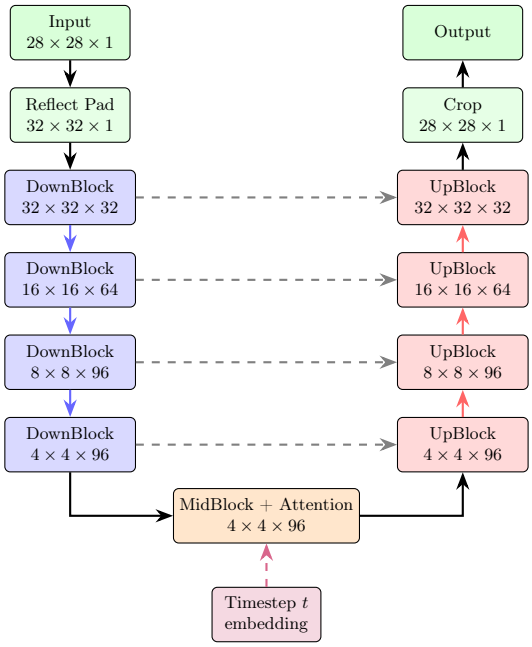


Fig. 1: UNet architecture for noise prediction.

Design Choices We deliberately kept the model small for MNIST. Table 1 summarizes the key parameters.

Table 1: UNet architecture configuration.

Parameter	**Value**
Base channels	32
Channel multipliers	$(1, 2, 3, 3)$
Block channels	$(32, 64, 96, 96)$
Resolution levels	$32 \to 16 \to 8 \to 4$
Self-attention	Only at 4×4 bottleneck
Total parameters	$\sim$2.7M

A few things worth noting:

– **Parameter reduction:** The original diffusers default configuration (multipliers $(1, 2, 4, 4)$, 2 layers per block, attention at multiple resolutions) gives around 6.7M parameters - overkill for 28×28 MNIST. We trimmed it to 2.7M by reducing multipliers to $(1, 2, 3, 3)$, using 1 layer per block, and restricting attention to the bottleneck only.

- **Reflect padding:** MNIST images are 28×28, but four levels of $2\times$ downsampling need a dimension divisible by $2^4 = 16$. We pad to 32×32 using reflect mode (rather than zero-padding) to avoid boundary artifacts, then crop back after the UNet forward pass.
- **Attention placement:** Self-attention is expensive and most useful at low resolutions where the receptive field matters. At 4×4, each spatial position can attend to all 16 positions cheaply. Putting attention at 32×32 would be wasteful for simple digits.

2.2 DDPM: Training and Sampling

The Forward Process DDPM [2] defines a forward (noising) process that gradually corrupts a clean image x_0 into Gaussian noise over $T = 1000$ timesteps:

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

which can be sampled in closed form as:

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{2}$$

The cumulative signal retention $\bar{\alpha}_t$ is determined by the noise schedule. We use the **cosine schedule** [7]:

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \qquad f(t) = \cos^2\left(\frac{t/T + s}{1 + s} \cdot \frac{\pi}{2}\right), \qquad s = 0.008 \tag{3}$$

The cosine schedule is smoother than the original linear schedule and avoids the issue of too-rapid noise injection at early timesteps.

Training Objective The network ϵ_θ is trained to predict the noise that was added:

$$\mathcal{L} = \mathbb{E}_{t, x_0, \epsilon}\Big[\|\epsilon - \epsilon_\theta(x_t, t)\|^2\Big] \tag{4}$$

Each training step samples a random $t \sim \text{Uniform}(0, T{-}1)$, noises x_0 via Eq. 2, and computes the MSE between the true and predicted noise. Super simple - no adversarial training, no reconstruction loss, just noise prediction.

Reverse Process (Sampling) To generate an image, we start from $x_T \sim \mathcal{N}(0, \mathbf{I})$ and apply the learned reverse step for $t = T{-}1, \ldots, 0$:

$$\hat{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\, x_t - \sqrt{\frac{1}{\bar{\alpha}_t} - 1}\ \epsilon_\theta(x_t, t) \tag{5}$$

$$\mu_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\, \beta_t}{1 - \bar{\alpha}_t}\, \hat{x}_0 + \frac{\sqrt{\alpha_t}\, (1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t}\, x_t \tag{6}$$

$$x_{t-1} = \mu_t + \sqrt{\tilde{\beta}_t}\, z, \qquad z \sim \mathcal{N}(0, \mathbf{I}) \text{ for } t > 0 \tag{7}$$

where $\tilde{\beta}_t = \frac{\beta_t (1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t}$ is the posterior variance and $\hat{x}_0$ is clipped to $[-1, 1]$ for stability.

Training Setup Table 2 lists our training hyperparameters.

Parameter	**Value**
Epochs	100
Optimizer	AdamW (weight decay 10^{-4})
Learning rate	10^{-3}
LR scheduler	CosineAnnealingLR ($\eta_{\min} = 10^{-6}$)
Batch size	512
EMA decay	0.995
Timesteps (T)	1000
Beta schedule	Cosine ($s = 0.008$)
Mixed precision	float16 (CUDA)

Table 2: Training configuration.

Exponential Moving Average (EMA) with decay 0.995 is a relatively simple addition to the pipeline, which aims at reducing noise noise in model weight updates, and, empirically, it made our diffusion model generate better images. The entire training run took $\approx$ 13 minutes on RTX 5090, converging to a final MSE of 0.0356 (train) / 0.0357 (eval).

Training Results Figure 2 shows the training and evaluation loss over 100 epochs. The loss drops quickly in the first 10 epochs and then slowly converges. The tight train/eval gap suggests we're not overfitting, which makes sense - MNIST has 60K training images and our model is relatively small.

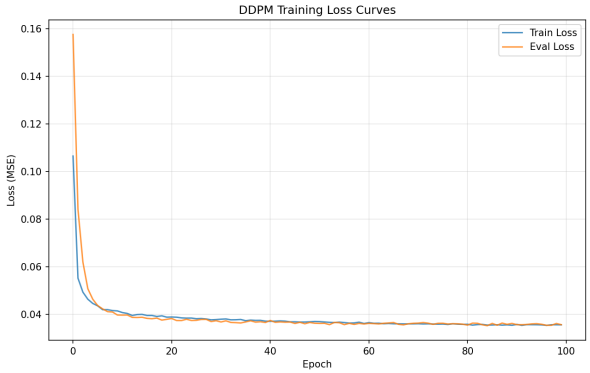
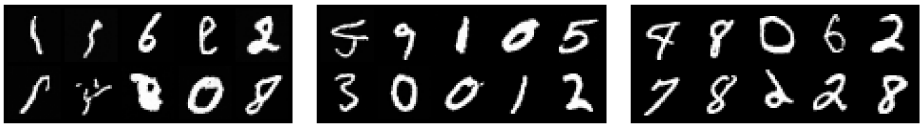


Fig. 2: Training and evaluation loss (MSE) over 100 epochs

Figure 3 shows sample quality at different stages of training. By epoch 10, digits are recognizable but noisy. By epoch 50, most samples look reasonable. Epoch 100 shows the final quality - most digits are clear, though some remain ambiguous.



(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 3: Generated samples at different training stages

DDPM Samples Figure 4 shows 16 samples generated with full 1000-step DDPM sampling. We can see that the digits are quite sharp, and basically unambiguous.

Figure 5 shows the denoising trajectory for a single sample. Structure emerges around $t \approx 500$ and sharpens progressively.

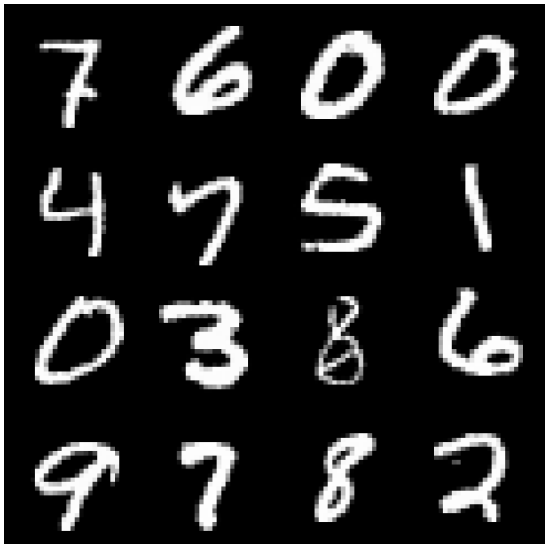


Fig. 4: DDPM images (1000 steps)

Fig. 5: DDPM denoising progression

Each sample takes approximately **9.98 seconds** to generate, since DDPM requires a full pass through all 1000 timesteps. This is the main practical limitation of DDPM - it's slow.

2.3 DDIM: Faster Inference

DDIM [10] offers a solution to DDPM's slow sampling: it defines a *non-Markovian* reverse process that can skip timesteps. The key insight is that the same trained noise prediction network ϵ_θ can be reused with a completely different sampling formula - no retraining needed.

Sampling Formula Given a subsequence of S timesteps uniformly sampled from $\{0, 1, \ldots, T-1\}$, the DDIM update rule is:

$$x_{\tau_{i-1}} = \sqrt{\bar{\alpha}_{\tau_{i-1}}}\,\hat{x}_0 + \sqrt{1 - \bar{\alpha}_{\tau_{i-1}} - \sigma^2}\,\epsilon_\theta(x_{\tau_i}, \tau_i) + \sigma\, z \tag{8}$$

where $\hat{x}_0$ is reconstructed the same way as in DDPM (Eq. 5), and the noise scale σ is controlled by a parameter η:

$$\sigma_{\tau_i} = \eta \cdot \sqrt{\frac{1 - \bar{\alpha}_{\tau_{i-1}}}{1 - \bar{\alpha}_{\tau_i}}} \cdot \sqrt{1 - \frac{\bar{\alpha}_{\tau_i}}{\bar{\alpha}_{\tau_{i-1}}}} \tag{9}$$

When $\eta = 0$, sampling is fully **deterministic** - no random noise is added at any step. This is what we use, following the DDIM paper's finding that $\eta = 0$ gives the best sample quality. When $\eta = 1$, DDIM reduces to DDPM.

Implementation Our DDIM sampler is a thin wrapper around the trained DDPM model. It reads the DDPM's precomputed $\bar{\alpha}_t$ buffers directly and constructs a uniform subsequence of S steps from the original $T = 1000$. The final step uses a sentinel value $\tau_{-1} = -1$ with $\bar{\alpha}_{-1} = 1.0$, which means "no remaining noise" and produces a clean image.

DDIM Samples Figure 6 shows 16 samples generated with DDIM (100 steps, $\eta = 0$), and Figure 7 shows the denoising progression.

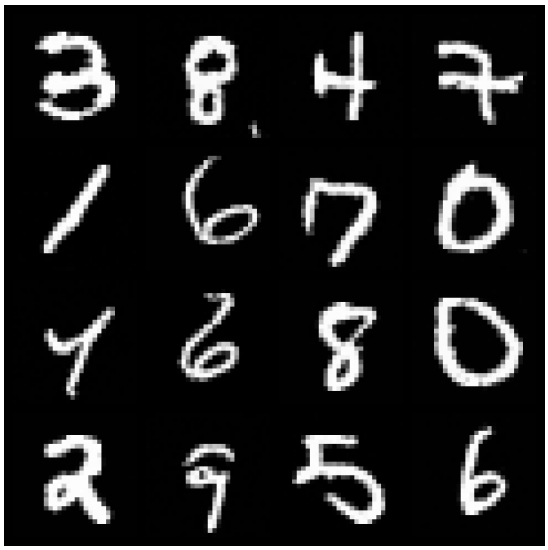


Fig. 6: Grid of 16 generated MNIST digits using DDIM (100 steps, $\eta = 0$). Quality is comparable, although a bit worse than DDPMs, but at a fraction of the cost.

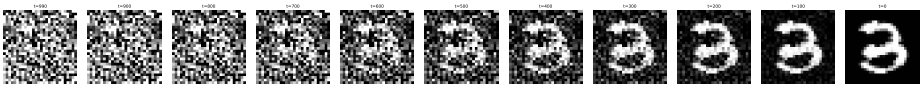


Fig. 7: DDIM denoising progression for a single sample (100 steps). Compared to DDPM (Figure 5), the trajectory looks similar but with larger jumps between snapshots since only 100 out of 1000 steps are used.

Speed Comparison Since we reduce the number of steps to just 10%, we get a significant improvement in generation speed.

Table 3: Generation speed comparison: DDPM vs DDIM.

Method	**Steps**	**Avg. time/sample**	**Speedup**
DDPM	1000	9.98 s	1×
DDIM (η=0)	100	1.05 s	9.5×

A **9.5× speedup** with no retraining and minimal quality loss is a great trade-off. The speedup is roughly proportional to the step reduction (1000/100 = 10×), with slight overhead from the DDIM formula being marginally more complex per step.

2.4 Distribution Evaluation

As a simple way to assess how close generated digits are to real MNIST, we used UMAP [6] to project both real and generated samples into 2D. Each 28×28 image is flattened to a 784-dimensional pixel vector, and UMAP reduces these to 2 components while preserving both local neighborhood structure and global cluster relationships.

We generated 1000 samples using DDIM (50 steps, $\eta = 0$) and compared them against 1000 real MNIST test set images. Figure 8 shows the result.

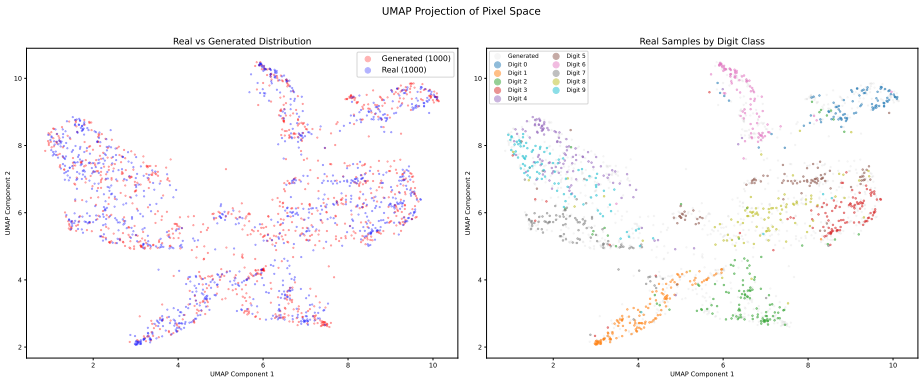


Fig. 8: UMAP projection of 1000 real vs 1000 generated MNIST samples in pixel space

The UMAP plot shows an overlap - generated samples (red) generally fall in the same regions as real samples (blue). The right panel reveals that real digits form reasonably distinct clusters by class (especially digits like 0 and 1), and generated samples fill similar regions. However, the overlap isn't perfect: some generated points lie in between clusters, corresponding to the ambiguous or malformed digits we see in the grids.

2.5 Discussion

What Worked Well

– **Training converged smoothly.** The loss curve shows clean, monotonic convergence with no instabilities or mode collapse - a common failure mode of GANs that diffusion models simply don't have. Train and eval loss tracked each other closely, meaning no overfitting.

- **EMA made a noticeable difference.** Samples from EMA-averaged weights were visibly sharper than from raw training weights. The 0.995 decay from the DDPM paper worked well out of the box.
- **DDIM just works.** Plugging in DDIM as a drop-in replacement for DDPM sampling - with no retraining - and getting a 9.5× speedup is remarkable. The deterministic ($\eta = 0$) mode produced clean results.
- **The cosine schedule is the right choice for MNIST.** The original linear schedule was designed for 256×256 images; the cosine schedule distributes noise more evenly across timesteps, which matters more for low-resolution images where even small amounts of noise destroy information quickly.

What Didn't Turn Out as Expected

- **Some generated digits are ambiguous.** Looking at the sample grids, most digits are recognizable, but a fraction look like they could be multiple different digits (is that a 3 or a 5? a 4 or a 9?).
- **Loss plateaus early.** The MSE loss essentially stops improving after epoch 50 or so (dropping from 0.037 to 0.036). More epochs beyond 100 would likely yield diminishing returns without architectural changes.

Why Results Might Not Be Great Several factors limit the quality of our generated samples:

- **Small model (2.7M parameters).** State-of-the-art diffusion models use 100M+ parameters. Our model has limited capacity to learn the full distribution of handwritten digit variations.
- **Pixel-space training.** We operate directly on raw pixels rather than in a learned latent space. Pixel-space models must learn both low-level texture details and high-level structure simultaneously, which might be less efficient than decoupling these with a VAE encoder.
- **Unconditional generation.** Without class conditioning, the model must allocate capacity to represent all 10 digit classes simultaneously. Classifier-free guidance [3] can significantly improve per-class quality.

3 Latent Diffusion

3.1 Introduction

The pixel-space diffusion model from Section 1 works, but it has a fundamental inefficiency: the UNet must simultaneously learn low-level texture details (small gradients of gray around edges, noise patches) and high-level structure (digit identity, global shape) from raw pixel values. Every denoising step operates on the full 28×28 resolution, even though most of the semantic content could be captured in a much smaller representation.

Latent Diffusion Models (LDMs) [8] address this by splitting the problem into two stages. First, a Variational Autoencoder (VAE) [4] learns to compress

images into a compact latent space — in our case, from $1 \times 28 \times 28$ (784 values) down to $2 \times 4 \times 4$ (32 values), a 24.5× compression ratio. Then a diffusion model operates entirely in this latent space, where each denoising step processes 32 values instead of 784. And we use UNet instead of a generic autoencoder because it does shape the latent space into something resembling a multivariate Gaussian.

The payoff is twofold: generation is faster because the UNet processes much smaller tensors, and the diffusion model can focus on learning the distribution of latent features rather than pixel-level details (those are already handled by the decoder).

3.2 Stage 1: Variational Autoencoder

VAE Architecture The VAE consists of an encoder that compresses images to a Gaussian latent distribution and a decoder that reconstructs images from latent samples. Both share the same building blocks — residual blocks, attention layers, and spatial resampling — but without timestep conditioning (the VAE has no notion of diffusion steps).

Figure 9 shows the overall structure.

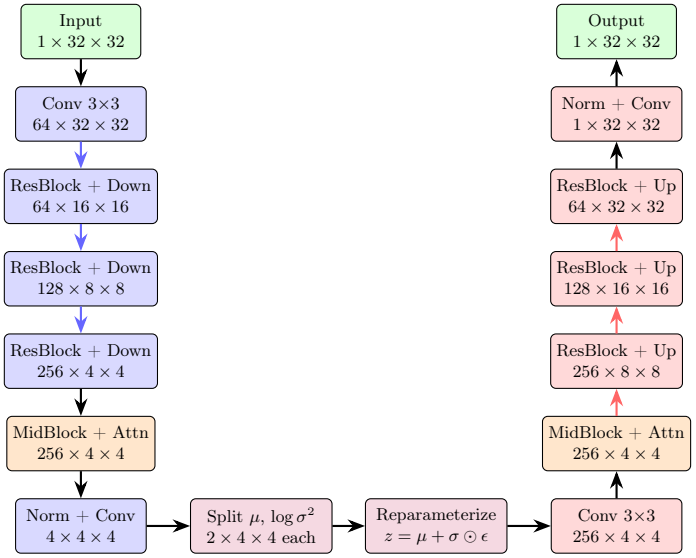


Fig. 9: VAE architecture.

The encoder (blue) compresses a padded 32×32 input through three downsampling levels to a 4×4 spatial resolution, then projects to μ and $\log \sigma^2$ (purple). The decoder (red) mirrors the encoder with upsampling. Self-attention is applied at the 4×4 bottleneck (orange) in both encoder and decoder.

Parameter	Value
Base channels	64
Channel multipliers	$(1, 2, 4)$
Block channels	$(64, 128, 256)$
Resolution levels	$32 \to 16 \to 8 \to 4$
Latent channels	2
Latent spatial size	4×4
Compression ratio	$784 \to 32$ (24.5×)

Table 4: VAE architecture configuration.

A few design notes:

- **No timestep conditioning.** Unlike the diffusion UNet, the VAE's residual blocks do not take a timestep embedding. The VAE is a standard autoencoder — it sees the clean image once and reconstructs it.
- **Compression.** Three levels of 2× downsampling take 32×32 to 4×4, and 1 input channel becomes 2 latent channels. This gives a 24.5× compression factor. Although it might look excessive at first, in the original paper they compressed $3 \times 512 \times 512$ images into $4 \times 64 \times 64$ vectors, achieving 48× compression factor.

Training Objective We use classic ELBO loss with a very small KL weight: $\beta = 10^{-5}$ (taken as something similar to Stable Diffusion). This is deliberate: for LDM, the primary goal is high-fidelity reconstruction, not a perfectly regularized latent space. A larger β would push the posterior closer to a standard normal but at the cost of blurrier reconstructions, which we want to avoid.

VAE Training Setup Table 5 lists the VAE training configuration.

Table 5: VAE training configuration.

Parameter	Value
Epochs	100
Optimizer	AdamW (weight decay 10^{-4})
Learning rate	10^{-4}
LR scheduler	CosineAnnealingLR ($\eta_{\min} = 10^{-6}$)
Batch size	512
KL weight (β)	10^{-5}
EMA decay	0.995
Gradient clipping	max norm 1.0
Mixed precision	float16 (CUDA)

Training took ≈ **5.9 minutes** on RTX 5090, converging to a final MSE of 0.0062 (train) / 0.0078 (eval).

VAE Training Results Figure 10 shows the loss curve over 100 epochs. The reconstruction loss drops sharply in the first 10 epochs and then gradually converges. The gap between train and eval loss is minimal, indicating no overfitting.

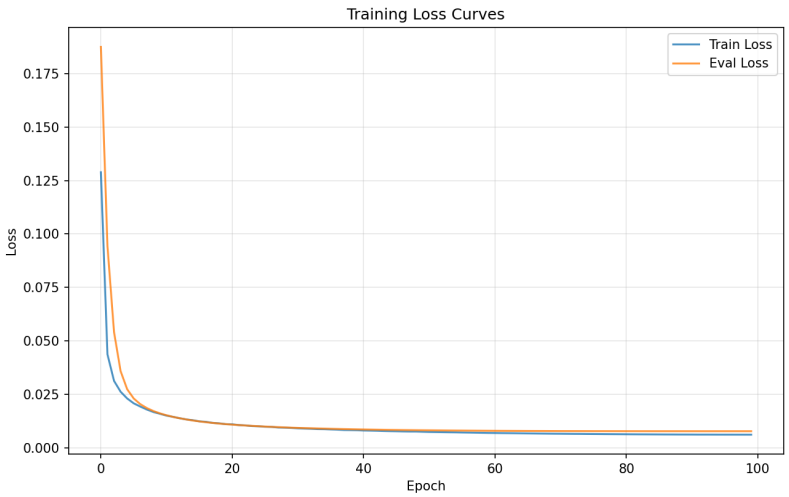


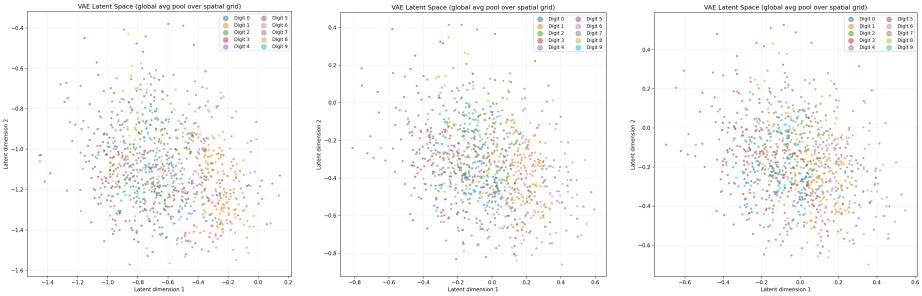
Fig. 10: VAE training and evaluation loss over 100 epochs

Figure 11 shows how reconstruction quality evolves during training, with good-enough digits being reconstructed at Epoch 10 already. However, I wouldn't run it for less than 50 epochs, since I encountered an issue of the whole LDM setup failing to generate digits with VAE trained for 20 epochs.

(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 11: VAE reconstruction quality at different training stages. Top rows: original MNIST digits. Bottom rows: VAE reconstructions

Figure 12 shows the learned latent space visualized via a 2D scatter plot of the encoded representations, colored by digit class. Despite using only 2 latent channels (and a very small KL weight), the latent space shows that there is some structure - different digit classes occupy distinct regions, with some expected overlap between visually similar digits (e.g., 3/5/8, 4/9).



(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 12: VAE latent space visualization at different training stages

3.3 Stage 2: Latent Diffusion

LDM Pipeline Overview With the VAE trained and frozen, we now train a diffusion model to generate latent features rather than raw pixels. The full generation pipeline is:

1. **Encode** (training only): $z_o = Enc(x)$.
2. **Diffuse**: Run the forward/reverse diffusion process on z_0 in latent space $(2 \times 4 \times 4)$, exactly as DDPM/DDIM operate on pixels - but on way smaller representation.
3. **Decode** (generation only): $\hat{x} = \text{Dec}(z_0)$, mapping latent features back to pixel space.

Two implementation details deserve attention:

Latent scaling factor. The VAE's latent distribution has an arbitrary variance that depends on training dynamics. If the latent variance is far from 1.0, the diffusion noise schedule (designed for unit-variance data) will be poorly calibrated. Following Rombach et al. [8] (Section 3.3), we normalize latents by a scaling factor $s = 1/\text{std}(z)$ computed over the training set. In our case, $s \approx 1.067$, meaning the raw latents already have near-unit variance - the cosine noise schedule transfers well. Although our model would work without it, we arrived at this option when debugging the LDM, and since it improves stability over other data, we decided to keep it.

Deterministic encoding. When encoding images for diffusion training, we use the posterior *mode* (mean) rather than a stochastic sample. This avoids injecting VAE reparameterization noise on top of the diffusion noise, which would effectively double the noise at each timestep and degrade training.

Latent UNet Architecture The latent UNet follows the same architecture as the pixel-space UNet from Section 1, but adapted for the much smaller latent tensors. Table 6 compares both.

Table 6: Pixel-space UNet vs. Latent UNet configuration.

Parameter	**Pixel UNet**	**Latent UNet**
Input size	$1 \times 32 \times 32$	$2 \times 8 \times 8$
Base channels	32	64
Channel multipliers	$(1, 2, 3, 3)$	$(1, 2)$
Block channels	$(32, 64, 96, 96)$	$(64, 128)$
Resolution levels	4	2
Layers per block	1	1
Attention levels	Bottleneck only	Second level only
Total parameters	$\sim$2.7M	$\sim$2.8M

We deliberately kept the UNet architecture identical to the pixel-space model - same building blocks, same training infrastructure — and only adjusted the configuration (channel widths, number of levels) to fit the latent tensor size. Despite having similar parameter counts, the two UNets have somewhat different structures: the pixel UNet distributes 2.7M parameters across 4 resolution levels with conservative channel widths, while the latent UNet concentrates 2.8M parameters across just 2 levels with wider channels. The latent UNet has fewer residual blocks (8 vs. 14) but compensates with more channels per block, giving each block more representational capacity.

We use only 2 resolution levels rather than 4 because the VAE already compresses images to 4×4 spatial resolution - further downsampling would produce a 2×2 bottleneck, where 3×3 convolutions see mostly zero-padding, GroupNorm operates on just 8 values per group, and self-attention over 4 tokens becomes trivial. To avoid this degenerate regime, we limit the UNet to a single downsampling step and pad the 4×4 latents to 8×8 using reflect padding (the same strategy as the pixel UNet's $28 \to 32$ padding). This means the UNet processes $2\times$ more spatial positions than strictly necessary, but at 8×8 the computational cost is negligible.

Latent Diffusion Training The diffusion training setup is nearly identical to Section 1 — same cosine noise schedule, same $T = 1000$ timesteps, same noise prediction objective:

$$\mathcal{L}_{\text{LDM}} = \mathbb{E}_{t, z_0, \epsilon}\Big[\|\epsilon - \epsilon_\theta(z_t, t)\|^2\Big] \tag{10}$$

The key difference from pixel-space training: we set `clip_denoised=False` during sampling, since latent values are not bounded to $[-1, 1]$ like pixel values.

Table 7 lists the training configuration.

Table 7: Latent diffusion training configuration.

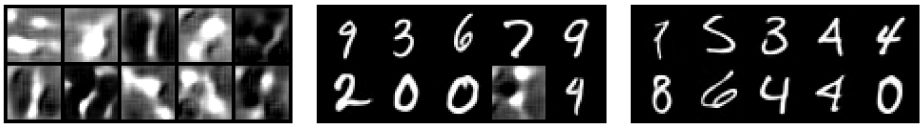
Parameter	**Value**
Epochs	100
Optimizer	AdamW (weight decay 10^{-4})
Learning rate	10^{-3}
LR scheduler	CosineAnnealingLR ($\eta_{\min} = 10^{-6}$)
Batch size	512
Timesteps (T)	1000
Beta schedule	Cosine ($s = 0.008$)
EMA decay	0.995
Gradient clipping	max norm 1.0
Mixed precision	float16 (CUDA)
Latent scaling factor	1.067
Clip denoised	False

Training took $\approx$ **8.9 minutes**, converging to a final MSE of 0.325 (train) / 0.334 (eval). Note that these loss values are not directly comparable to the pixel-space DDPM loss (0.036): the latent representation has a different scale and dimensionality, so the per-element MSE is on a different scale.

LDM Training Results Figure 13 shows the latent diffusion loss curve. The convergence pattern is similar to pixel-space DDPM: a rapid initial drop followed by a long, gradual convergence. The train/eval gap is slightly larger than in pixel-space training, but remains small.

Fig. 13: LDM loss

Figure 14 shows generated samples at different training stages. At epoch 10, the model produces blurry but occasionally recognizable digit shapes. By epoch 50, most samples are sharp and identifiable. Epoch 100 shows the final quality — digits are well-formed with clear strokes.



(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 14: LDM generated samples at different training stages

3.4 LDM Generation Results

DDPM Sampling in Latent Space Figure 15 shows 16 samples generated with full 1000-step DDPM sampling in latent space. The pipeline generates a $2 \times 4 \times 4$ latent code via DDPM, then decodes it through the frozen VAE to produce a 28×28 image.

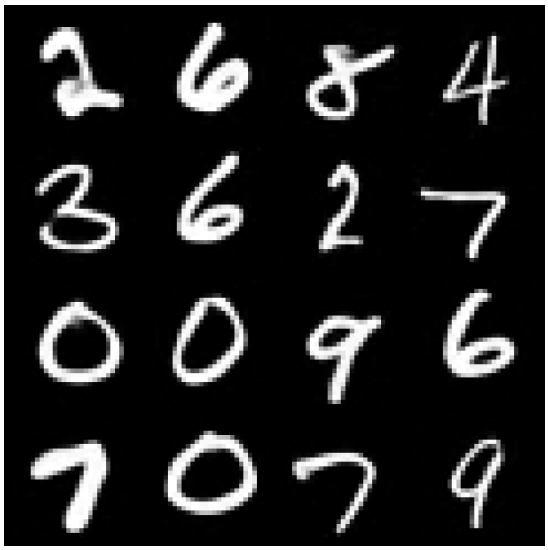


Fig. 15: LDM samples using DDPM (1000 steps) in latent space.

Figure 16 shows the denoising progression for a single sample. The visualization shows the decoded pixel-space image at several points during the reverse

diffusion process. Structure emerges in the latent space and becomes visible in pixel space as the denoising progresses.

Fig. 16: LDM DDPM denoising progression. Each panel shows the VAE-decoded image at a different point in the 1000-step reverse process.

Each LDM DDPM sample takes approximately **3.85 seconds**, compared to 9.98 seconds for pixel-space DDPM — a 2.6× speedup from operating on the compressed latent representation.

DDIM Sampling in Latent Space Just as in pixel space, DDIM [10] can be used as a drop-in replacement for DDPM sampling in latent space. We use 100 DDIM steps with $\eta = 0.05$ (nearly deterministic, with a small amount of stochasticity).

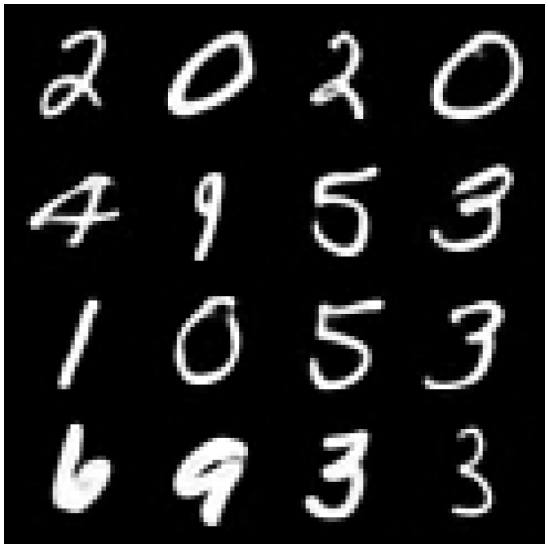


Fig. 17: LDM samples using DDIM (100 steps, $\eta = 0.05$) in latent space.

Fig. 18: LDM DDIM denoising progression for a single sample (100 steps).

Each LDM DDIM sample takes approximately **0.42 seconds** — a 9.2× speedup over LDM DDPM, consistent with the 10× step reduction.

3.5 Pixel-Space vs. Latent-Space Comparison

Generation Speed Table 8 brings together the generation timing results across all four method combinations. The latent-space approach consistently generates samples faster than pixel-space, and DDIM consistently accelerates both.

Method	**Space**	**Steps**	**Time/sample**	**Speedup**
DDPM	Pixel	1000	9.98 s	1×
DDIM (η=0)	Pixel	100	1.05 s	9.5×
DDPM	Latent	1000	3.85 s	2.6×
DDIM (η=0.05)	Latent	100	0.42 s	**23.8**×

Table 8: Generation speed comparison across all methods on RTX 5090.

Training Cost The LDM pipeline takes ∼15% longer to train overall, but the marginal cost would amortize well: once the VAE is trained and frozen, additional diffusion experiments (different architectures, hyperparameters, conditioning strategies) only need the 8.9-minute diffusion training, not the full 14.8 minutes.

Pipeline	**Training time**	**Parameters**
Pixel DDPM	12.9 min	2.7M (UNet)
LDM (VAE + Diffusion)	5.9 + 8.9 = 14.8 min	9.8M + 2.8M

Table 9: Training time comparison.

Sample Quality Both approaches produce recognizable MNIST digits, but the LDM samples appear noticeably **sharper** than the pixel-space equivalents. This is a somewhat surprising result given that the two UNets have similar parameter counts (∼2.7–2.8M). A likely explanation is that the diffusion model's

job is simply easier in latent space: it only needs to learn the distribution of 32-dimensional latent features (which the VAE has already organized into a smooth, structured manifold), rather than learning the full pixel-level distribution from scratch. The VAE decoder then handles the deterministic mapping from latent codes to sharp pixel images.

3.6 Discussion

What Worked Well

- **The two-stage approach is clean and efficient.** The VAE trains in under 6 minutes and provides a 24.5× compression. The frozen VAE then serves as a reusable infrastructure for diffusion experiments, decoupling perceptual compression from generative modeling.
- **Significant generation speedup.** Operating in a $2 \times 4 \times 4$ latent space instead of $1 \times 32 \times 32$ pixel space gives a consistent 2.5–2.6× speedup for both DDPM and DDIM. Combined with DDIM, the total speedup reaches 24× over the pixel DDPM baseline.
- **Sharper samples from latent diffusion.** Despite similar parameter counts, the LDM produces visually sharper digits than pixel-space diffusion. The VAE's learned latent space provides a better starting point for the diffusion model.
- **DDPM/DDIM interchangeability transfers to latent space.** The same trick from Section 1 — swapping DDPM for DDIM at inference time with no retraining — works identically in latent space, giving another ∼9× speedup.

What Didn't Turn Out as Expected

- **Total training time is longer, not shorter.** The LDM pipeline (VAE + diffusion) takes 14.8 minutes total vs. 12.9 minutes for pixel DDPM. The generation speedup, however, pays off if we generate many samples, or reuse the VAE across multiple experiments, which we did.
- **The UNet padding overhead is inelegant.** Our 4×4 latents get padded to 8×8 for the two-level UNet, meaning half the spatial positions are padding. The UNet must learn to ignore these padded regions, which wastes capacity. A single-level UNet or a VAE producing larger latents could avoid this.

Potential Improvements

1. **Class-conditional generation.** As with pixel-space diffusion, adding class conditioning with classifier-free guidance [3] would be the single biggest quality improvement, allowing targeted generation of specific digits.
2. **Quantitative evaluation.** As with Section 1, we rely on visual inspection. FID scores or classification accuracy of generated digits would provide objective comparison between pixel and latent approaches.

3. **KL weight annealing.** Our fixed $\beta = 10^{-5}$ produces good reconstructions but may not optimally structure the latent space. Annealing β from 0 to a target value during training could give better trade-offs between reconstruction quality and latent space regularity.

4 Classifier-Free Guidance

4.1 Introduction

The unconditional latent diffusion model from Section 2 generates plausible MNIST digits, but offers no control over *which* digit appears. Classifier-Free Guidance (CFG) [3] solves this by conditioning the noise prediction network on a class label, and then amplifying the conditional signal at sampling time.

During training, the model learns to predict noise both conditionally (given a class label) and unconditionally (label dropped with probability $p_{\text{uncond}} = 0.1$). At inference, the two predictions are combined:

$$\hat{\epsilon} = \epsilon_{\text{uncond}} + w \cdot (\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}) \tag{11}$$

where w is the **guidance scale**. At $w = 0$ the model ignores the class label entirely; at $w = 1$ it uses standard conditional predictions; at $w > 1$ it amplifies the class signal beyond what was learned during training, trading diversity for class adherence.

We implemented two approaches for injecting the class label into the UNet, described next.

4.2 Conditioning Approaches

Channel Concatenation Following Ho & Salimans [3], each class label is mapped to a learnable spatial pattern of size $1 \times 4 \times 4$ via an embedding table, then concatenated to the feature tensor as an extra input channel. The UNet therefore takes 3 input channels (2 latent + 1 conditioning) and predicts 2 output channels.

This approach is simple and adds minimal overhead, but the conditioning only enters at the input layer - the model must propagate class information through the entire network.

Cross-Attention Following Rombach et al. [8], each class label is mapped to a dense 128-dimensional embedding vector. The UNet's self-attention layers are augmented with cross-attention blocks where queries come from UNet features and keys/values come from the class embedding. The conditioning enters deep inside the network at every attention level rather than only at the input. This generalizes naturally to richer conditioning signals (e.g. text token sequences) but adds extra parameters and computation.

4.3 Training

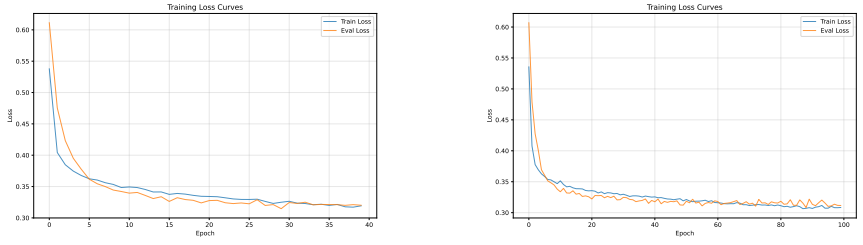
Both models build on the same frozen VAE and latent UNet architecture from Section 2. The only differences are the conditioning mechanism and the training duration. Table 10 lists the configurations. Channel concat was trained for just 40 epochs, since at this point it already achieved great generation, and the losses plateaued.

Parameter	**Channel Concat**	**Cross-Attention**
Epochs	40	100
Training time	5.3 min	15.2 min
Batch size	1024	1024
Unconditional dropout	10%	10%
Cross-attention dim	-	128
Final train loss	0.319	0.308
Final eval loss	0.320	0.312

Table 10: Training configuration for both conditioned models.

All other hyperparameters are shared with the unconditioned LDM from Section 2: AdamW (lr $= 10^{-3}$, weight decay 10^{-4}), cosine LR schedule, EMA decay 0.995, gradient clipping at 1.0, cosine noise schedule with $T = 1000$, and the same latent scaling factor $s \approx 1.067$.

Figure 19 shows the loss curves. Both models converge smoothly, with the cross-attention model reaching a slightly lower final loss after its longer training run.



(a) Channel concatenation (40 epochs) (b) Cross-attention (100 epochs)

Fig. 19: Training loss curves for both conditioned models.

4.4 Generation Results

Both models are sampled with DDIM (100 steps, $\eta = 0.05$) and a default guidance scale of $w = 3.0$. Figure 20 shows one sample per class from each model. Both produce recognizable, class-correct digits at this guidance strength.

(a) Channel concatenation (b) Cross-attention

Fig. 20: Class-conditional samples at $w = 3.0$ (one per digit, 0–9).

One practical difference is sampling speed. Because the CFG wrapper performs two forward passes per step (conditional + unconditional), conditioned generation is roughly 2× slower than unconditioned LDM. Cross-attention adds further overhead from the extra attention computation.

Model	Time / sample	vs. uncond. LDM
Unconditioned LDM (DDIM 100)	0.42 s	1×
Channel concat + CFG	0.80 s	1.9×
Cross-attention + CFG	1.05 s	2.5×

Table 11: Per-sample generation time (DDIM, 100 steps, $\eta = 0.05$).

4.5 Effect of Guidance Scale

The guidance scale w controls the quality–diversity trade-off. To visualize this, we generated 10 samples for each digit class at 10 different w values: 0.0, 0.5, 1.0, 2.0, 3.0, 5.0, 7.0, 10.0, 50.0, and 100.0. All samples for a given class start from the same initial noise, so differences across columns are purely due to the guidance strength.

Figures 21 and 22 show the full grids for both models (rows = digit classes, columns = w values, 10 samples per cell).

Fig. 21: CFG sweep for channel concatenation model.

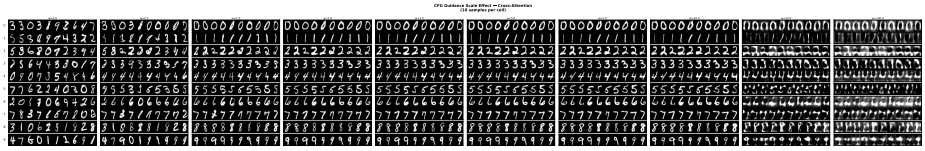


Fig. 22: CFG sweep for cross-attention model.

The pattern is consistent across both models and reveals four distinct regimes:

- $w < 1$ **(under-guided):** The model essentially ignores the class label. Samples are random digits that frequently disagree with the requested class. At $w = 0$ the output is purely unconditional.
- $w \approx 1$–2 **(weakly guided):** Class adherence improves noticeably, but occasional misclassifications persist - some "3" rows still contain stray 9s or 5s.
- $w \approx 3$–7 **(sweet spot):** Digits are sharp, class-correct, and still retain reasonable intra-class diversity (varying stroke widths, slant, etc.). $w = 3$ is a good default.
- $w \geq 10$ **(over-guided):** Samples become increasingly homogeneous - the 10 samples per cell look more alike. At extreme values ($w = 50$–100), the over-amplified guidance signal produces pixel blobs that resemble identical digit-like structures rather than sharp digits.

4.6 Discussion

What Worked Well

- **Both conditioning approaches produce good results.** Channel concatenation and cross-attention generate visually similar quality at $w = 3$, despite their very different mechanisms. For simple tasks like MNIST class labels, the simpler channel-concat approach is sufficient.
- **The guidance scale provides a smooth quality–diversity dial.** Moving w from 1 to 5 smoothly transitions from diverse-but-noisy to sharp-and-consistent, giving practical control over the desired output style.
- **DDIM sampling integrates cleanly.** The same DDIM sampler from Section 2 works unchanged - CFG only modifies the noise prediction, not the sampling algorithm.

What Didn't Turn Out as Expected

- **DDPM sampling is incompatible with CFG in latent space.** Full 1000-step DDPM sampling with the CFG wrapper produced poor results - the guidance bias accumulates over many stochastic steps, degrading the output. We switched to DDIM-only generation (100 steps, $\eta = 0.05$), which avoids this issue. Dynamic thresholding (clamping the predicted $\hat{x}_0$ at each step) was attempted but did not help.

- **Extreme guidance scales cause mode collapse.** At $w = 50$–100, instead of producing ultra-sharp digits, the model generates pixelated blobs. The over-amplified guidance pushes predictions far outside the training distribution, where the network's behavior is undefined.
- **Cross-attention is slower than concat.** The extra attention computation adds an overhead (1.05 s vs. 0.80 s per sample), without a clear quality advantage on this 10-class task. Cross-attention's benefits would likely be more apparent with richer conditioning (e.g. text descriptions).

Potential Improvements

1. **Richer conditioning.** Cross-attention naturally extends to multi-token conditioning (e.g. "a bold handwritten seven"). This might work for a differently-labeled MNIST.
2. **Negative prompting.** Using a different unconditional baseline (e.g. conditioning on a different class as the "negative" direction) could provide finer control over generation.

5 Rectified Flow

5.1 Introduction

The diffusion models from Sections 1–3 all rely on a *discrete* noise schedule: the forward process adds noise according to a predefined $\bar{\alpha}_t$ schedule over $T = 1000$ integer timesteps, and sampling must reverse this same schedule step by step. Rectified Flow [5] offers a fundamentally simpler alternative. Instead of a complex noise schedule, Rectified Flow defines a *straight-line* interpolation between data and noise in continuous time $t \in [0, 1]$, and trains a network to predict the velocity along this path.

The key differences from DDPM are:

- **Linear interpolation** replaces the noise-schedule-based forward process.
- **Velocity prediction** replaces noise prediction - the network learns the direction from data to noise rather than the noise itself.
- **ODE sampling** replaces the stochastic reverse process - generation is a deterministic integration from $t = 1$ (noise) to $t = 0$ (data) with far fewer steps.

The same UNet architecture from Section 1 is reused without modification; only the loss function and sampling procedure change.

5.2 Method

Forward Interpolation and Velocity Prediction Given a clean image x_0 and Gaussian noise $\epsilon \sim \mathcal{N}(0, \mathbf{I})$, Rectified Flow defines the forward interpolation as a straight line:

$$x_t = (1 - t)\, x_0 + t\, \epsilon, \qquad t \in [0, 1] \tag{12}$$

At $t = 0$ we have the clean image; at $t = 1$, pure noise. The velocity along this path is the constant vector:

$$v = \epsilon - x_0 \tag{13}$$

The training objective is to predict this velocity:

$$\mathcal{L} = \mathbb{E}_{t,x_0,\epsilon}\Big[\|v_\theta(x_t, t) - (\epsilon - x_0)\|^2\Big] \tag{14}$$

This is conceptually simpler than DDPM's loss: there is no noise schedule, no $\bar{\alpha}_t$ coefficients, and no posterior variance computation.

ODE Sampling To generate an image, we start from $x_1 \sim \mathcal{N}(0, \mathbf{I})$ and solve the ODE $\mathrm{d}x/\mathrm{d}t = -v_\theta(x_t, t)$ from $t = 1$ to $t = 0$. The simplest integrator is Euler's method with uniform step size $\Delta t = 1/N$:

$$x_{t-\Delta t} = x_t - \Delta t \cdot v_\theta(x_t, t) \tag{15}$$

With $N = 50$ steps, each sample requires only 50 forward passes through the UNet - compared to 1000 for DDPM and 100 for DDIM. Higher-order integrators such as the midpoint (Heun) method can improve accuracy at the cost of extra model evaluations per step, and logit-normal time sampling [1] can concentrate training signal near $t = 0.5$ where the velocity field changes fastest. We use the basic Euler integrator with uniform time sampling in all experiments reported here.

5.3 Training

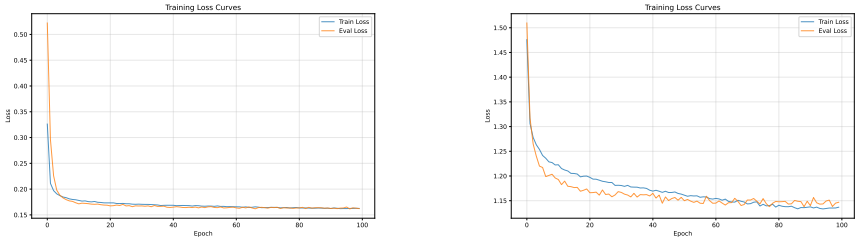
We trained Rectified Flow models in both pixel space and latent space (reusing the VAE from Section 2). Both use the same UNet backbone and optimizer settings as the corresponding DDPM models, differing only in the loss function and time sampling. Table 12 lists the configurations.

Parameter	**Pixel Rectified Flow**	**Latent Rectified Flow**
Epochs	100	100
Training time	13.1 min	10.7 min
Optimizer	AdamW (10^{-3}, wd 10^{-4})	AdamW (10^{-3}, wd 10^{-4})
Batch size	512	512
EMA decay	0.995	0.995
Euler steps	50	50
Base channels	32	64
Channel multipliers	$(1, 2, 3, 3)$	$(1, 2)$
Time sampling	Uniform	Uniform
Final train loss	0.163	1.137
Final eval loss	0.162	1.147

Table 12: Rectified Flow training configuration.

The latent Rectified Flow operates on the same $2 \times 4 \times 4$ latent space as Section 2's LDM, with scaling factor $s \approx 1.067$. Training times are comparable to DDPM (13.1 min for pixel Rectified Flow vs. 12.9 min for DDPM), since the per-epoch cost is dominated by UNet forward passes, which are identical in both frameworks.

Figure 23 shows the loss curves. Both models converge smoothly with tight train/eval gaps. The pixel Rectified Flow loss drops rapidly in the first 20 epochs (from 0.33 to 0.17) and then gradually settles around 0.163. The latent Rectified Flow follows a similar pattern, converging from 1.48 to 1.14.



(a) Pixel-space Rectified Flow (b) Latent-space Rectified Flow

Fig. 23: Training and evaluation loss over 100 epochs.

Note that the Rectified Flow velocity loss (Eq. 14) and DDPM's noise-prediction loss (Section 1, Eq. 4) are *not* directly comparable - they predict different targets with different scales.

Figures 24 and 25 show how sample quality evolves during training.

(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 24: Pixel-space Rectified Flow samples at different training stages.

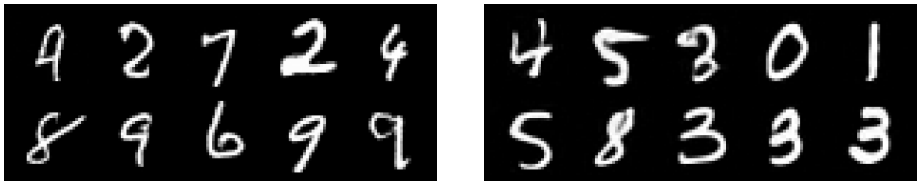
(a) Epoch 10 (b) Epoch 50 (c) Epoch 100

Fig. 25: Latent-space Rectified Flow samples at different training stages.

Digits are already clearly-recognisable at epoch 10, with high-quality examples being generated at epoch 100. Suprisingly, the Latent-space model manages to produce visually better digits. I assume it might be the case because most of the noise is being removed by the autoencoder, letting the model concentrate on recreating the actual digit.

5.4 Generation Results

Figure 26 shows 10 final samples from each model. The pixel-space Rectified Flow generates recognizable digits with good variety, though some samples are blurrier than DDPM's output. The latent Rectified Flow produces noisier samples, with rare artifacts from the VAE reconstruction (i.e. digit 3 in the top row here).



(a) Pixel-space Rectified Flow (b) Latent-space Rectified Flow

Fig. 26: Final samples (Euler, 50 steps).

Figure 27 shows the ODE integration trajectory for a single pixel-space sample. Unlike DDPM's stochastic denoising where structure emerges gradually around $t \approx 500$, the Rectified Flow trajectory shows a smooth, continuous transformation from noise to digit.

Fig. 27: Pixel-space Rectified Flow denoising progression (Euler, 50 steps).

5.5 Comparison with Diffusion Models

The main advantage of Rectified Flow is inference speed. Table 13 compares sampling times (**during training runs**) across all methods on the same hardware (RTX 5090).

Model	Steps	Time (10 samples)	Speedup
DDPM (pixel)	1000	5.51 s	1×
LDM + DDIM (latent)	100	1.0 s	2.5×
Rectified Flow pixel (Euler)	50	0.26 s	21×
Rectified Flow latent (Euler)	50	0.18 s	31×

Table 13: Sampling speed comparison across all generative approaches.

Rectified Flow achieves a 21–31× speedup over DDPM while requiring the same training time. The latent Rectified Flow is the fastest overall at 0.018 s per sample - fast enough for real-time applications.

However, the speed gain comes at a quality cost. While Rectified Flow samples are recognizable, they are generally blurrier and less sharp than DDPM's output. This is expected: with only 50 first-order Euler steps, the ODE integration introduces truncation error that accumulates along the trajectory. DDPM's 1000 small stochastic steps, combined with an optimized noise schedule, provide finer control over the denoising process.

Training stability is comparable between the two approaches. Both converge smoothly with no signs of mode collapse or training instability (Figures 2 and 23). The main difference is in the loss magnitudes, which are not directly comparable because DDPM predicts noise ϵ while Rectified Flow predicts velocity $v = \epsilon - x_0$.

5.6 Discussion

What Worked Well

- **Dramatic inference speedup.** At 21–31× faster than DDPM, Rectified Flow makes generation nearly instant. The 50 Euler steps versus 1000 DDPM steps is the single biggest practical advantage.
- **Framework simplicity.** No noise schedule to tune, no posterior variance computation, no $\bar{\alpha}_t$ bookkeeping. The entire forward process is one line: $x_t = (1-t)x_0 + t\epsilon$.
- **Architecture reuse.** The same UNet from Section 1 works without modification. The only change is replacing noise prediction with velocity prediction and integer timesteps with continuous time (scaled appropriately for the sinusoidal position embedding).
- **Stable, predictable training.** Loss curves are smooth and monotonically decreasing with tight train/eval gaps, similar to DDPM. No mode collapse, no instabilities.

What Didn't Turn Out as Expected

- **Lower sample quality than DDPM.** Despite comparable training time, Rectified Flow samples are noticeably blurrier than DDPM's. The digits are recognizable but lack the sharpness of DDPM's 1000-step output - some strokes are fuzzy and edges are less defined.

Potential Improvements

1. **Logit-normal time sampling.** Uniform time sampling treats all timesteps equally, but the velocity field is most complex near $t = 0.5$. Logit-normal sampling [1] would concentrate training signal in this critical region, potentially improving the learned velocity field without changing inference.
2. **Higher-order ODE solvers.** A midpoint (Heun) sampler would achieve second-order accuracy, potentially matching the quality of 100 Euler steps with only 50 midpoint steps (at the cost of 2 model evaluations per step).
3. **More sampling steps.** Increasing from 50 to 200 Euler steps would reduce truncation error at the cost of proportionally slower inference (still $5\times$ faster than DDPM).
4. **Reflow.** Liu et al. [5] propose iteratively straightening the learned trajectories by re-training on the model's own samples. Straighter trajectories allow fewer steps with less error.

6 Conclusions

In this lab, I implemented different versions of the diffusion and Rectified Flow models. The most satisfying results were obtained from latent CFG diffusion models - the digits looked sharp, artifacts were minimal and the generation was fast. I believe that, conceptually, the main reason behind this was the actual condifitoning, which directed the generation process towards the distribution of expected class-digits.

In terms of implementation, the lab took 30+ hours (including going through relevant literature), and some money spent on the Vast AI compute. Although Claude was used to prepare base versions of the models and the template of the reports, those were later verified and modified manually. A suite of unit tests was also implemented to avoid accidentally 'vibe-coding' away of things that already work. Finally, a number of scripts were written to easily setup the repository, and run all kinds of generation and sampling tasks.

In terms of the potential improvements, I believe that the grading system of that lab can be improved. Personally, it took me the most efforts to set up the models in latent space, with early versions stubbornly producing pixelated blobs. And yet, based on the grading schema, it is the least 'important' part of the assignment.

However, overall, after completing the lab I feel way more comfortable working with both Diffusion and Flow models, which were the main reason of me enrolling in the course in the first place.

The scope of this report covers all the main assignments. Finishing them, I was already behind the deadline, so I decided against diving into the bonus tasks this time.

References

1. Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., et al.: Scaling rectified flow transformers for high-resolution image synthesis. In: International Conference on Machine Learning (2024)
2. Ho, J., Jain, A., Abbeel, P.: Denoising diffusion probabilistic models. Advances in Neural Information Processing Systems **33**, 6840–6851 (2020)
3. Ho, J., Salimans, T.: Classifier-free diffusion guidance. arXiv preprint arXiv:2207.12598 (2022)
4. Kingma, D.P., Welling, M.: Auto-encoding variational Bayes. arXiv preprint arXiv:1312.6114 (2013)
5. Liu, X., Gong, C., Liu, Q.: Flow straight and fast: Learning to generate and transfer data with rectified flows. arXiv preprint arXiv:2209.03003 (2022)
6. McInnes, L., Healy, J., Melville, J.: Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426 (2018)
7. Nichol, A.Q., Dhariwal, P.: Improved denoising diffusion probabilistic models. International Conference on Machine Learning pp. 8162–8171 (2021)
8. Rombach, R., Blattmann, A., Lorenz, D., Esser, P., Ommer, B.: High-resolution image synthesis with latent diffusion models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 10684–10695 (2022)
9. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. In: Medical Image Computing and Computer-Assisted Intervention. pp. 234–241 (2015)
10. Song, J., Meng, C., Ermon, S.: Denoising diffusion implicit models. arXiv preprint arXiv:2010.02502 (2020)