Exam project – Probabilistic Machine Learning

Benjamin Baadsager npr151@alumni.ku.dk

Christian Clasen nsw337@alumni.ku.dk

Yasin Baysal cmv882@alumni.ku.dk

January 16, 2026

1 Introduction

This report presents our work on the final project in Probabilistic Machine Learning, divided into two main parts. In Part A, we study diffusion-based generative models by extending a DDPM baseline on MNIST and analyzing how modeling choices affect training and sample quality. In Part B, we investigate Gaussian Process regression with noisy inputs, comparing a standard GP baseline to extensions that explicitly model input uncertainty through theoretical and empirical analysis[1].

2 Part A: Diffusion-based generative models

Diffusion-based generative models generate data by reversing a stochastic noising process. In DDPMs, a forward Markov chain adds Gaussian noise to data $x_0 \sim q(x_0)$, with transitions $q(x_t \mid x_{t-1}) = \mathcal{N}\big(x_t \mid \sqrt{1-\beta_t}\, x_{t-1}, \beta_t I\big)$ under a fixed variance schedule $\{\beta_t\}_{t=1}^T$. As T increases, $q(x_T)$ approaches a standard Gaussian. The model learns reverse transitions $p_\theta(x_{t-1} \mid x_t)$ via denoising score matching [2]. Our baseline uses a time-conditioned U-Net with Gaussian Fourier features, trained with an ELBO implemented as an MSE noise-prediction loss. Sampling iteratively applies the learned reverse diffusion steps from Gaussian noise. Variations of this model are explored next (code changes available in Appendix 4.2).

2.1 Direction 1: Converting the MNIST DDPM template to Flow Matching

First, we reinterpret the DDPM training template using Flow Matching (FM), which learns a continuous-time velocity field transporting samples from a source to a data distribution, enabling direct comparison with stochastic diffusion sampling while keeping architecture and dataset fixed [3]. Given source p and target q, FM considers a probability path $(p_t)_{t\in[0,1]}$ with $p_0 = p$ and $p_1 = q$. A velocity field $u_t^\theta \colon \mathbb{R}^d \to \mathbb{R}^d$ defines a flow ψ_t via $\frac{d}{dt}\psi_t(x) = u_t(\psi_t(x))$, with $\psi_0(x) = x$. Learning the marginal velocity $u_t(x)$ directly is intractable, so FM uses conditional paths $p_{t|1}(x \mid x_1)$ with $x_1 \sim q$. We employ the conditional optimal-transport path $p_{t|1}(x \mid x_1) = \mathcal{N}(x \mid tx_1, (1-t)^2 I)$, obtained by sampling $X_0 \sim \mathcal{N}(0, I)$, $X_1 \sim q$, and setting $X_t = (1-t)X_0 + tX_1$. The conditional velocity is $u_t(x \mid x_1) = X_1 - X_0$, so that (see Appendix 4.1)

$$\mathcal{L}_{\text{CFM}}^{\text{OT,Gauss}}(\theta) = \mathbb{E}_{X_0,X_1,t}\left[\left\|u_t^\theta(X_t) - (X_1 - X_0)\right\|^2\right], \quad t \sim \mathcal{U}(0,1),\ X_0 \sim \mathcal{N}(0,I),\ X_1 \sim q.$$

Unlike DDPMs, which parameterize via score functions and sample stochastically, Flow Matching explicitly regresses velocity fields and samples deterministically via ODE integration. Thus, we replace the DDPM noise-prediction objective with this velocity regression loss while reusing the same U-Net architecture without making any modifications.We observe several failure modes: too few ODE steps (< 20) cause discretization artifacts, insufficient training yields noisy samples, and overly small learning rates lead to slow convergence and underfitting (see Appendix 4.3.1).

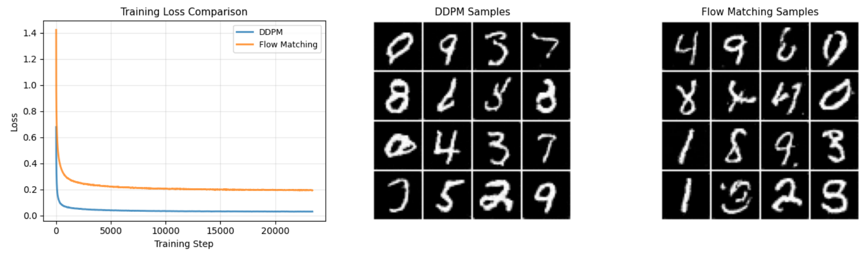


Figure 1: Comparison between DDPM and Flow Matching on MNIST. Left: training loss curves. Middle: samples generated by the DDPM model. Right: samples generated by the Flow Matching model.

Figure 1 compares training dynamics and samples from DDPM and Flow Matching. Although the training losses are not directly comparable, both models converge stably and generate coherent MNIST digits with comparable visual quality. Quantitative evaluation using nearest-neighbor distances and FID-like metrics is deferred to Section 2.5.

[1]All code for this project is available at `https://github.com/yabay0/ProbMLProject`.

2.2 Direction 2: Combining an Autoencoder with a Diffusion Model

Latent diffusion models reduce computational cost by applying the diffusion process in a lower-dimensional learned latent space rather than pixel space. High-dimensional pixel representations contain redundancy that an autoencoder can compress, and performing diffusion in this compact space drastically reduces memory and training time. In this direction, we train a convolutional autoencoder with encoder $E\colon \mathbb{R}^{28\times 28} \to \mathbb{R}^d$ and decoder $D\colon \mathbb{R}^d \to \mathbb{R}^{28\times 28}$ (with $d = 32$) using reconstruction loss $\mathcal{L}_{\text{AE}} = \mathbb{E}_{x\sim q}\|x - D(E(x))\|^2$ (see Appendix 4.1.2). After pretraining, we freeze the autoencoder and train a diffusion model $p_\theta(z_{t-1} \mid z_t)$ in the latent space $z = E(x)$. Sampling generates $z_0 \sim p_\theta$ via reverse diffusion, then decodes to pixel space as $x = D(z_0)$. This two-stage approach mirrors Stable Diffusion's architecture at a smaller scale [4]. We observe two main failure modes in latent diffusion. Using too few reverse-diffusion steps leads to incomplete or faint samples, while mismatched initial noise scales in latent space cause decoding collapse. Proper step counts and latent normalization are therefore important for stable generation during the training (see failure modes in Appendix 4.3.2).

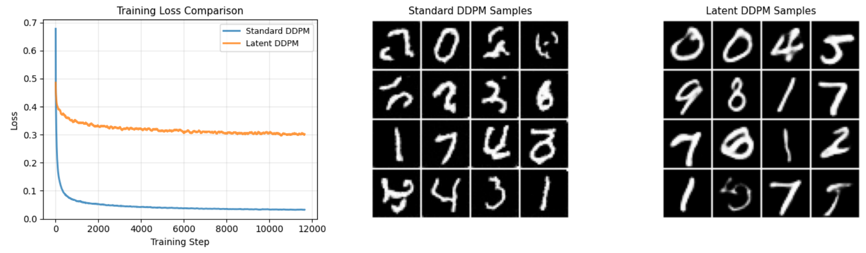


Figure 2: Comparison of standard DDPM and latent DDPM on MNIST. Left: training loss curves. Middle: samples generated by standard pixel-space DDPM. Right: samples generated by the latent-space DDPM using pretrained autoencoder.

While the standard DDPM achieves lower training loss in Figure 2, the latent DDPM produces visually more coherent digits with reduced computational cost. This highlights the trade-off between training efficiency and reconstruction fidelity.

2.3 Direction 3: Continuous-Time SDE Diffusion Model

Next, we reinterpret the discrete-time DDPM as a continuous-time SDE following [5] and [6], unifying diffusion, score-based, and flow-based models. The forward process is $dx = f(x,t)\,dt + g(t)\,dw$, and for the variance-preserving (VP) schedule, $f(x,t) = -\frac{1}{2}\beta(t)x$ and $g(t) = \sqrt{\beta(t)}$ (see Appendix 4.1.3). The model learns the score $s_\theta(x,t) \approx \nabla_x \log p_t(x)$ via denoising score matching. Sampling proceeds stochastically using the reverse SDE or deterministically via the probability flow ODE, $dx = \left[f(x,t) - \frac{1}{2}g(t)^2\nabla_x \log p_t(x)\right]dt$, enabling tradeoffs between speed and diversity. We implement VP-SDE on MNIST using the same U-Net as the baseline. Both Euler-Maruyama (SDE) and Euler (ODE) integration produce samples similar to discrete DDPM. Failure modes include numerical instability from overly aggressive $\beta(t)$ schedules and sensitivity to the initial noise scale during ODE sampling (see failure modes in Appendix 4.3.3).

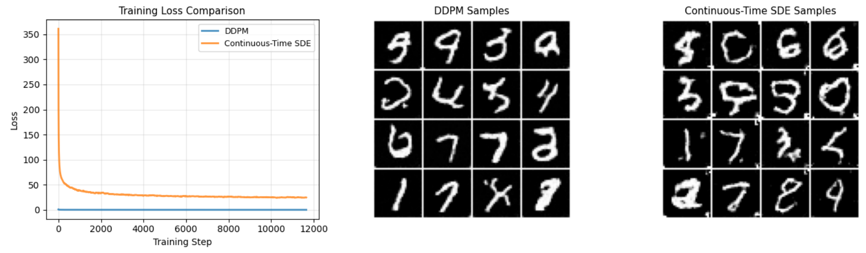


Figure 3: Comparison of discrete DDPM and continuous VP-SDE on MNIST. Left: training loss curves. Middle: DDPM samples. Right: VP-SDE samples generated with the probability flow ODE.

Both models show stable training dynamics, though the continuous-time SDE converges more slowly and to a higher loss level than the discrete DDPM. While the VP-SDE produces recognizable digits, its samples are generally noisier and less sharp than those from the DDPM, indicating reduced sample fidelity under this continuous-time formulation.

2.4 Direction 4: Classifier-Free Guidance in Diffusion Models

Lastly, we implement classifier-free guidance (CFG) following [1] to enable class-conditional generation without an auxiliary classifier. A single diffusion model is trained for conditional and unconditional generation by randomly dropping class labels with probability p_{uncond}. At inference, conditional and unconditional noise predictions are combined as

$$\tilde{\epsilon}_\theta(x_t, t, c) = (1 + w)\epsilon_\theta(x_t, t, c) - w\,\epsilon_\theta(x_t, t, \emptyset),$$

where w controls guidance strength (see Appendix 4.1.4). We extend the baseline U-Net with learned class embeddings concatenated with time embeddings, while keeping the diffusion schedule unchanged. Training with $p_{\text{uncond}} = 0.1$ yields stable convergence. At $w = 0$, the model behaves as an unconditional DDPM with diverse samples. Moderate guidance improves class consistency and sharpness, while larger values reduce diversity. Failure modes include excessive guidance (large w) and insufficient unconditional training, preventing effective guidance (see Appendix 4.3.4).

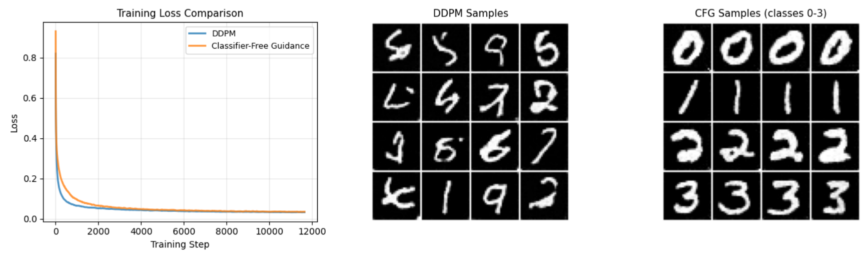


Figure 4: Comparison of DDPM and classifier-free guidance on MNIST. Left: training loss curves. Middle: unconditional DDPM samples. Right: CFG samples conditioned on digit classes 0–3 with guidance scale $w = 2.0$.

Figure 4 shows that classifier-free guidance achieves training dynamics comparable to the unconditional DDPM while enabling controlled, class-conditional generation with improved sharpness compared to the standard DDPM.

2.5 Quantitative Comparisons

Table 1 compares all model variants to the baseline DDPM using metrics capturing sample quality, diversity, and distributional alignment. Inception Score (IS) is computed from a separately trained MNIST classifier and reflects both sample quality and class diversity, while Fréchet Inception Distance (FID) measures the distance between real and generated feature distributions extracted from the same classifier. Sample quality is measured as the mean classifier confidence on generated images, and diversity is quantified by the entropy of the predicted class distribution. Final training loss is reported for reference only, as losses are not directly comparable across model families. All models are trained with identical diffusion horizons ($T = 1000$), optimizers, learning rates, and evaluation sample sizes. Metric definitions are provided in Appendix 4.1.5, while code explanations are given in Appendix 4.2.5.

Model	IS ↑	FID ↓	Quality ↑	Diversity ↑	Final Loss ↓
Baseline DDPM	7.318 ± 0.297	93.43	0.908 ± 0.151	2.272	0.0429
Direction 1: Flow Matching	7.457 ± 0.234	63.84	0.906 ± 0.160	2.296	0.1818
Direction 2: Latent Diffusion	8.190 ± 0.217	41.83	0.941 ± 0.129	2.295	0.2791
Direction 3: VP-SDE	7.345 ± 0.263	67.49	0.906 ± 0.154	2.276	21.1732
Direction 4: Classifier-Free Guidance	1.029 ± 0.038	55.05	0.996 ± 0.032	2.302	0.0250

Table 1: Quantitative comparison of generative models on MNIST. Arrows indicate whether higher or lower values are better.

Relative to the baseline DDPM, Flow Matching (Direction 1) achieves similar IS and diversity but improves FID, reflecting its deterministic transport-based formulation despite a higher regression loss. Latent diffusion (Direction 2) yields the best FID and quality scores, consistent with denoising in a compact latent space, while preserving diversity at the cost of additional reconstruction error. The VP-SDE model (Direction 3) performs comparably in IS and diversity but shows degraded FID and unstable losses, likely due to numerical sensitivity in the continuous-time integration. Classifier-free guidance (Direction 4) achieves the highest sample quality and lowest FID by explicitly biasing generation toward class-consistent modes, but this induces a strong fidelity–diversity trade-off, as reflected by its low IS. Overall, the results highlight how different modeling choices trade diversity, fidelity, and training stability relative to the baseline DDPM.

3 Part B: Function Fitting with Noisy Inputs

We consider a regression dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, where inputs and outputs are noisy. The data-generating process is

$$y_i = f(x_i - \Delta_i) + \varepsilon_i, \qquad \Delta_i \sim \mathcal{N}(0, \sigma_x^2), \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_y^2),$$

with $\sigma_x^2 = 0.01$ and $\sigma_y^2 = 0.0025$. The ground truth function is $f(x) = -x^2 + \frac{2}{1+\exp(-10x)}$.

B.1 Fitting a standard Gaussian Process

Kernel Choice, Parameters, and Optimization

In Section B.1, the input noise is ignored and the observed inputs are treated as noise-free. We model the latent function f as a Gaussian Process (GP) prior $f \sim \mathcal{GP}(0, k_\theta(\cdot,\cdot))$, using the squared exponential (RBF) kernel

$$k_\theta(x, x') = \sigma_f^2 \exp\left(-\frac{(x-x')^2}{2\ell^2}\right), \tag{1}$$

which is twice continuously differentiable and therefore suitable for the subsequent noisy-input model. The kernel hyperparameters are $\theta = (\ell, \sigma_f^2)$, while the observation noise variance σ_y^2 is either fixed or learned depending on the model variant.

Let $K \in \mathbb{R}^{n\times n}$ with entries $K_{ij} = k_\theta(x_i, x_j)$. Under $y = f(X) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, \sigma_y^2 I)$, the marginal likelihood is

$$p(y \mid X, \theta, \sigma_y^2) = \mathcal{N}\big(y;\, 0,\, K + \sigma_y^2 I\big),$$

with corresponding log marginal likelihood

$$\log p(y \mid X, \theta, \sigma_y^2) = -\tfrac{1}{2} y^\top (K + \sigma_y^2 I)^{-1} y - \tfrac{1}{2}\log|K + \sigma_y^2 I| - \tfrac{n}{2}\log(2\pi). \tag{2}$$

We maximize (2) with respect to the hyperparameters in log-parameter space. In practice, (2) is evaluated using a Cholesky factorization of $K + \sigma_y^2 I$ and triangular solves, with a small jitter term added to the diagonal to ensure numerical stability. To mitigate sensitivity to local optima, we verify that a coarse grid search yields solutions consistent with a local L-BFGS-B optimizer (see Appendix 5.1). We consider three baseline models: (a) noisy inputs $X = x_i$ with σ_y^2 learned from data, (b) noisy inputs $X = x_i$ with $\sigma_y^2 = 0.0025$ fixed, and (c) true inputs $X = x_i^{\text{Truth}} = x_i - \Delta_i$ with $\sigma_y^2 = 0.0025$ fixed.

Posterior over The Latent Function

On $X_* \subset [-1, 1]$ of $n_* = 100$ equally spaced points, the posterior over the latent function values $f_* = f(X_*)$ is Gaussian,

$$\begin{aligned} p(f_* \mid X_*, X, y, \theta, \sigma_y^2) &= \mathcal{N}(\mu_*, \Sigma_*), \\ \mu_* &= K_*^\top (K + \sigma_y^2 I)^{-1} y, \\ \Sigma_* &= K_{**} - K_*^\top (K + \sigma_y^2 I)^{-1} K_*, \end{aligned}$$

where $K_* = K(X, X_*)$ and $K_{**} = K(X_*, X_*)$. This follows from the joint Gaussianity induced by the GP prior and standard conditioning of multivariate Gaussians. Figure 5 shows the posterior mean μ_*, the 95% confidence interval $\mu_* \pm 1.96\sqrt{\operatorname{diag}(\Sigma_*)}$, and the ground truth function $f(x)$. Since we infer the latent function f rather than predictive observations y, the predictive variance excludes σ_y^2. For model (c), data are plotted against the true inputs x_i^{Truth}.

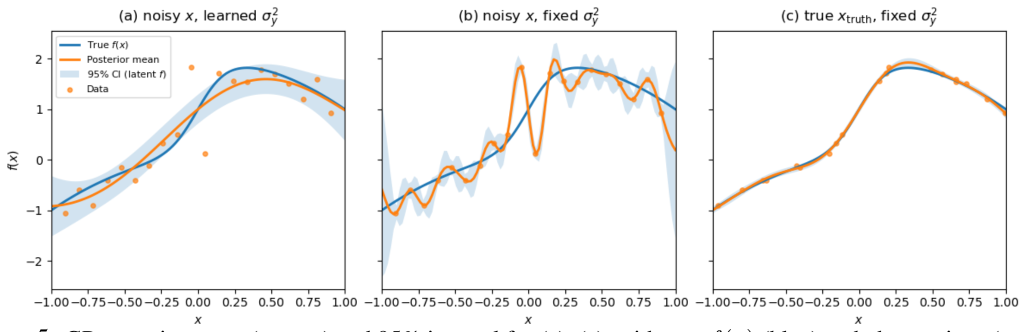


Figure 5: GP posterior mean (orange) and 95% interval for (a)–(c), with true $f(x)$ (blue) and observations (points).

Optimized Parameters and Discussion

The optimized hyperparameters are reported in Table 2, with $\widehat{\sigma}_y^2$ additionally reported for model (a).

Model	$\hat{\ell}$	$\widehat{\sigma_f^2}$	$\widehat{\sigma_y^2}$
(a) noisy x, σ_y^2 learned	0.7215	1.2090	0.1590
(b) noisy x, $\sigma_y^2 = 0.0025$	0.0788	1.1000	0.0025
(c) true x^{Truth}, $\sigma_y^2 = 0.0025$	0.3658	0.9298	0.0025

Table 2: Optimized kernel and noise variances for the baseline GP models.

Model (b), which fixes σ_y^2 while ignoring input noise, exhibits a very small length-scale ($\hat{\ell} \approx 0.08$), leading to an over-flexible posterior that attempts to explain input-induced variability as signal. Allowing σ_y^2 to vary in model (a) instead results in a substantially inflated noise estimate, absorbing variability caused by noisy inputs and yielding a smoother posterior with wider uncertainty bands. The signal variance σ_f^2, which controls the overall vertical scale of the latent function, remains of comparable magnitude across the three models, indicating that the dominant compensatory effects under input-noise misspecification are expressed primarily through the length-scale ℓ and the noise variance σ_y^2. Finally, model (c), which uses the true inputs and the correct noise level, achieves the best agreement with the ground truth function and well-calibrated uncertainty. These results highlight the limitations of standard GP regression under input noise.

B.2 Modeling Gaussian Process Regression with Noisy Inputs

We now explicitly model uncertainty in the inputs. For small input perturbations Δ_i, a first-order Taylor approximation $f(x_i - \Delta_i) \approx f(x_i) - \Delta_i f'(x_i)$ leads to the observation model

$$y = f(X) - Df'(X) + \varepsilon, \qquad D = \operatorname{diag}(\Delta),\ \ \varepsilon \sim \mathcal{N}(0, \sigma_y^2 I). \tag{3}$$

Conditional Noisy-Input GP Model (Taylor Approximation)

In this section, the input perturbations Δ are treated as known and fixed. We reuse the squared exponential (RBF) kernel (1). Since the noisy-input model involves the derivative process f', the kernel must be twice continuously differentiable in order for $\operatorname{Cov}(f'(x), f'(x'))$ to be well-defined. The required kernel derivatives are

$$\frac{\partial k(x,x')}{\partial x} = k(x,x')\frac{x'-x}{\ell^2}, \qquad \frac{\partial^2 k(x,x')}{\partial x \partial x'} = k(x,x')\left(\frac{1}{\ell^2} - \frac{(x-x')^2}{\ell^4}\right)$$

The correctness of the derivative process f' was verified via a prior sanity check based on sampling and numerical differentiation (Appendix 5.2). Let $f = [f(x_1), \ldots, f(x_n)]^\top$ and $f' = [f'(x_1), \ldots, f'(x_n)]^\top$. Under the GP prior, the joint distribution of function values and derivatives at the training inputs is

$$\begin{bmatrix} f \\ f' \end{bmatrix} \sim \mathcal{N}\!\left(0, \begin{bmatrix} K & K_1^\top \\ K_1 & K_2 \end{bmatrix}\right),$$

where $K_{ij} = k(x_i, x_j)$, $(K_1)_{ij} = \partial k(x_i, x_j)/\partial x_i$ and $(K_2)_{ij} = \partial^2 k(x_i, x_j)/(\partial x_i\, \partial x_j)$. Defining $z = [f^\top, (f')^\top]^\top$ and the linear map $A = [I \quad -D]$, (3) can be written as $y = Az + \varepsilon$. Conditioned on input perturbations Δ, observations have

$$\mathbb{E}[y \mid X, \Delta] = 0, \qquad \operatorname{Cov}(y \mid X, \Delta) = A \operatorname{Cov}(z)\, A^\top + \sigma_y^2 I,$$

yielding the conditional marginal likelihood given Δ, where $C_\Delta = \operatorname{Cov}(y \mid X, \Delta)$,

$$p(y \mid X, \Delta, \theta, \sigma_y^2) = \mathcal{N}\big(y;\, 0,\, C_\Delta\big), \qquad C_\Delta = K - DK_1 - K_1^\top D + DK_2 D + \sigma_y^2 I.$$

Hyperparameter Estimation and Conditional Posterior

Using the realizations of the input perturbations Δ, the kernel hyperparameters $\theta = (\ell, \sigma_f^2)$ and the observation noise variance σ_y^2 are estimated by maximizing the marginal log-likelihood $\log \mathcal{N}(y; 0, C_\Delta)$, yielding $\hat{\ell} = 0.3004$, $\widehat{\sigma_f^2} = 0.7382$, and $\widehat{\sigma_y^2} = 0.0046$ for the noisy-input GP model. For predictions, we set $\Delta_* = 0$, i.e., we predict the latent function at nominal test locations rather than noisy observations. Let $X_* \subset [-1, 1]$ be a grid of $n_* = 100$ equally spaced points and define $K_* = K(X, X_*)$ and $K_{**} = K(X_*, X_*)$ as earlier. Let $G_* = \frac{\partial}{\partial X} K(X, X_*)$ denote the derivative of the kernel with respect to the training inputs. Then $\operatorname{Cov}(f_*, z) = [K_*^\top,\ G_*^\top]$, so the cross-covariance between latent test function values $f_* = f(X_*)$ and observations y is

$$\operatorname{Cov}(f_*, y) = \operatorname{Cov}(f_*, Az) = [K_*^\top,\ G_*^\top]A^\top = K_*^\top - G_*^\top D.$$

The conditional posterior over the latent function values is then

$$p(f_* \mid X_*, X, y, \Delta, \theta, \sigma_y^2) = \mathcal{N}(\mu_*, \Sigma_*)$$

with $\mu_* = \operatorname{Cov}(f_*, y)\, C_\Delta^{-1} y$ and $\Sigma_* = K_{**} - \operatorname{Cov}(f_*, y)\, C_\Delta^{-1} \operatorname{Cov}(y, f_*)$.

Bayesian Inference over Input Noise

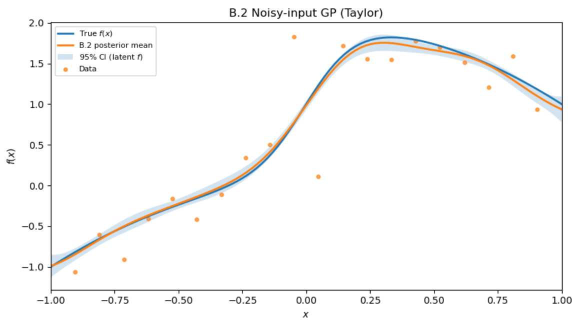
The conditional model above assumes that the Δ's are known. In practice, these perturbations are latent and should be inferred from the data. To account for input uncertainty, we thus also sample from the posterior $p(\Delta \mid X, y, \theta, \sigma_y^2, \sigma_x^2)$ using NUTS (with 4 chains) with the fitted (θ, σ_y^2) from the previous section and using the known σ_x^2. Given posterior samples $\{\Delta^{(s)}\}_{s=1}^S$, the marginal posterior predictive distribution of the latent function is obtained by Monte Carlo marginalization,

$$p(f_* \mid X_*, X, y) = \int p(f_* \mid X_*, X, y, \Delta)\, p(\Delta \mid X, y)\, \mathrm{d}\Delta \;\approx\; \frac{1}{S}\sum_{s=1}^{S} p(f_* \mid X_*, X, y, \Delta^{(s)}).$$

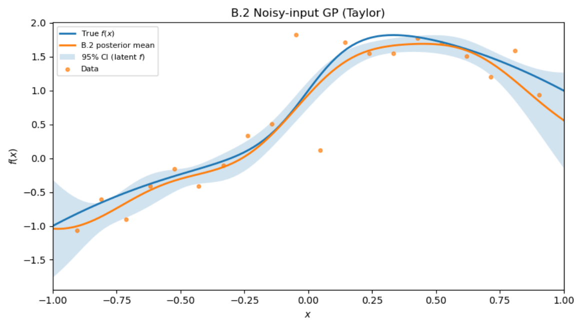
Let $p(f_* \mid X_*, X, y, \Delta^{(s)}) = \mathcal{N}(\mu_*^{(s)}, \Sigma_*^{(s)})$ denote the conditional GP posterior for a given sample $\Delta^{(s)}$. The marginal posterior predictive mean and variance are then estimated as

$$\mathbb{E}[f_* \mid X_*, X, y] \approx \frac{1}{S}\sum_{s=1}^{S} \mu_*^{(s)}, \qquad \mathrm{Var}(f_* \mid X_*, X, y) \approx \frac{1}{S}\sum_{s=1}^{S} \mathrm{diag}(\Sigma_*^{(s)}) + \mathrm{Var}_s(\mu_*^{(s)}),$$

corresponding to the law of total variance. Figure 6 shows the posterior mean and 95% confidence interval on the grid X_*, along with the ground truth $f(X_*)$ and training data. The left panel displays the conditional posterior $p(f_* \mid X_*, X, y, \Delta)$, while the right panel shows the marginal posterior $p(f_* \mid X_*, X, y)$ obtained via Monte Carlo marginalization over Δ. Marginalization increases posterior uncertainty in regions, where the input perturbations are weakly identified.



(a) Conditional posterior $p(f_* \mid X_*, X, y, \Delta)$.



(b) Marginal posterior $p(f_* \mid X_*, X, y)$.

Figure 6: Comparison of conditional and marginal posterior predictive distributions for the noisy-input GP.

Figure 7 shows posterior samples of $(\Delta_{10}, \Delta_{11})$, which concentrate around the true values $(-0.25, 0.25)$, indicating that the noisy-input GP model is able to recover the dominant input perturbations. Further implementation details and diagnostics are provided in Appendix 5.3.

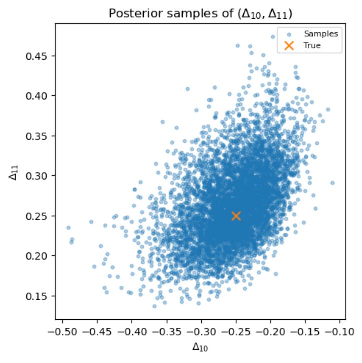


Figure 7: Posterior samples of $(\Delta_{10}, \Delta_{11})$ obtained using NUTS. The red cross indicates the true values $(-0.25, 0.25)$.

Comparison to B.1 and practical considerations

Compared to the standard GP baselines in Section B.1 that treats the observed inputs as noise-free, the noisy-input GP explicitly models input uncertainty via a first-order Taylor expansion and f', avoiding the failure modes observed in B.1: with fixed σ_y^2 the standard GP tends to overfit by driving the length-scale ℓ to very small values, while with learned σ_y^2 it absorbs input-induced variability into an inflated noise variance.

By contrast, the noisy-input model attributes this variability to input perturbations, yielding smoother posterior means and more realistically calibrated uncertainty. The main drawbacks are increased computational cost and dependence on modeling assumptions, in particular the validity of the first-order approximation and the need for scalable inference over the latent input errors Δ in practical applications. To make the noisy-input GP usable in practice, the model must be combined with scalable approximate inference schemes and either higher-order or non-perturbative treatments of input uncertainty, as the first-order Taylor approximation and full Bayesian inference over Δ do not scale well to large datasets.

References

[1] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. 2022. URL: https://arxiv.org/abs/2207.12598.

[2] Oswin Krause. *Probabilistic Machine Learning: Lecture Notes*. 2022.

[3] Yaron Lipman et al. *Flow Matching Guide and Code*. 2024. URL: https://arxiv.org/abs/2412.06264.

[4] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. URL: https://arxiv.org/abs/2112.10752.

[5] Yang Song. *Generative Modeling by Estimating Gradients of the Data Distribution*. 2021. URL: https://yang-song.net/blog/2021/score/ (visited on 01/14/2026).

[6] Yang Song et al. "Score-Based Generative Modeling through Stochastic Differential Equations". In: 2021. URL: https://openreview.net/forum?id=PxTIG12RRHS.

4 Appendix for Part A

4.1 Theoretical Sections for Directions in Part A

4.1.1 Direction 1: Derivation of the Conditional Velocity Field for the Linear Gaussian Path

This section provides the derivation underlying Direction 1 of Part A, following the Flow Matching literature in [3].

Let the source distribution be $p := p_0 = \mathcal{N}(0, I)$ and the target (data) distribution be $q := p_1$. Fix a data point $X_1 = x_1$. The linear Gaussian (conditional optimal transport) path is defined by the conditional distributions

$$p_{t|1}(x \mid x_1) = \mathcal{N}\big(x \mid tx_1, (1-t)^2 I\big), \quad t \in [0,1].$$

A random variable $X_{t|1} \sim p_{t|1}(\cdot \mid x_1)$ can be constructed by sampling $X_0 \sim \mathcal{N}(0, I)$ and setting

$$X_{t|1} = (1-t)X_0 + tx_1. \tag{4}$$

Differentiating 4 with respect to t yields

$$\frac{d}{dt}X_{t|1} = x_1 - X_0. \tag{5}$$

By definition, the conditional velocity field generating the conditional path is

$$u_t(X_{t|1} \mid x_1) := \frac{d}{dt}X_{t|1}.$$

To express this velocity as a function of the state x, solve 4 for X_0, such that

$$X_0 = \frac{x - tx_1}{1-t}, \quad x = X_{t|1},\ t < 1.$$

Substituting into 5 gives the closed-form conditional velocity field

$$u_t(x \mid x_1) = \frac{x_1 - x}{1-t}.$$

When X_t is sampled via 5, this further simplifies to

$$u_t(X_t \mid X_1) = X_1 - X_0,$$

which leads directly to the simplified conditional Flow Matching objective used in Direction 1 given by

$$\mathcal{L}_{\text{CFM}}^{\text{OT,Gauss}}(\theta) = \mathbb{E}_{t,X_0,X_1}\left[\left\|u_t^\theta(X_t) - (X_1 - X_0)\right\|^2\right], \quad X_t = (1-t)X_0 + tX_1.$$

This derivation justifies the training target used when converting the MNIST DDPM template to Flow Matching.

4.1.2 Direction 2: Latent Diffusion via a Pretrained Autoencoder

In Direction 2, we combine a pretrained autoencoder with a diffusion model trained in the learned latent space. The approach follows the central idea of Latent Diffusion Models (LDMs), i.e. replacing pixel-space diffusion with diffusion on a lower-dimensional representation that preserves perceptually relevant information while reducing computational cost [4].

Autoencoding and reconstruction objective. Let the data distribution be $q(x)$ on images $x \in \mathbb{R}^{28\times 28}$. We train a deterministic autoencoder (E, D) with encoder $E : \mathbb{R}^{28\times 28} \to \mathbb{R}^d$ and decoder $D : \mathbb{R}^d \to \mathbb{R}^{28\times 28}$, using

$$\mathcal{L}_{\text{AE}} = \mathbb{E}_{x\sim q}\left[\|x - D(E(x))\|_2^2\right].$$

After training, we freeze (E, D) and define the empirical latent distribution induced by the encoder as

$$q_E(z) := (E_{\#}q)(z) \quad \text{meaning} \quad z = E(x),\ x \sim q(x).$$

Here, q_E is the distribution of encoded training examples.

DDPM in latent space. We train a diffusion model in latent space with latent variable $z \in \mathbb{R}^d$. The forward noising process is defined analogously to a DDPM, but applied to z, such that

$$q(z_t \mid z_{t-1}) = \mathcal{N}\Big(z_t \mid \sqrt{1-\beta_t}\, z_{t-1},\ \beta_t I\Big), \qquad t = 1, \dots, T,$$

where $\{\beta_t\}$ is a fixed variance schedule and $q(z_0) = q_E(z)$. The marginal noising distribution has the form

$$q(z_t \mid z_0) = \mathcal{N}\big(z_t \mid \sqrt{\bar\alpha_t}\, z_0,\ (1-\bar\alpha_t)I\big), \quad \bar\alpha_t := \prod_{s=1}^{t}(1-\beta_s).$$

Reverse model and training objective. As in standard DDPMs, we parameterize the reverse process via a neural noise predictor $\epsilon_\theta(z_t, t)$ (implemented by a time-conditioned network). Using the usual DDPM reparameterization

$$z_t = \sqrt{\bar\alpha_t}\, z_0 + \sqrt{1-\bar\alpha_t}\,\epsilon, \qquad \epsilon \sim \mathcal{N}(0, I),$$

the simplified ELBO/denoising objective becomes the same MSE loss but in latent space:

$$\mathcal{L}_{\text{LDDPM}}(\theta) = \mathbb{E}_{z_0\sim q_E,\ \epsilon\sim\mathcal{N}(0,I),\ t}\Big[\|\epsilon - \epsilon_\theta(z_t, t)\|_2^2\Big], \qquad t \sim \text{Unif}\{1, \dots, T\}.$$

This is the pixel-space DDPM loss with x replaced by $z = E(x)$, matching the LDM perspective that diffusion can be trained on an encoded representation rather than raw pixels [4].

Sampling and decoding. Sampling proceeds by reverse diffusion in latent space with

$$z_T \sim \mathcal{N}(0, I), \qquad z_{t-1} \sim p_\theta(z_{t-1} \mid z_t),\ \ t = T, \dots, 1,$$

and we map the final latent back to pixel space via

$$\hat{x} = D(z_0).$$

4.1.3 Direction 3: Continuous-Time VP-SDE and Probability Flow ODE

This section derives the continuous-time diffusion formulation used in Direction 3 based on the theory in [6]. Here, we treat diffusion as a forward SDE that continuously perturbs data into noise, and generate samples by solving either the corresponding reverse-time SDE (stochastic) or its associated probability flow ODE (deterministic).

Forward SDE (variance preserving). Let $x(t) \in \mathbb{R}^d$ denote the data state at continuous time $t \in [0, T]$ with initial condition $x(0) \sim p_0$ (the data distribution). A general forward diffusion has the form

$$dx = f(x, t)\, dt + g(t)\, dw, \tag{6}$$

where w is standard Brownian motion. In the variance-preserving (VP) SDE parameterization, we set

$$f(x, t) = -\tfrac{1}{2}\beta(t)\, x, \qquad g(t) = \sqrt{\beta(t)},$$

for a monotonically increasing noise schedule $\beta(t)$.

Reverse-time SDE (stochastic sampling). A key result is that any diffusion SDE (6) admits a reverse-time SDE that transports noise back to data. If $p_t(x)$ denotes the marginal density of $x(t)$, then the reverse-time dynamics are

$$dx = \left[f(x, t) - g(t)^2 \nabla_x \log p_t(x)\right] dt + g(t)\, d\bar{w}, \tag{7}$$

where $\bar{w}$ is Brownian motion in reverse time. Therefore, sampling reduces to estimating the time-dependent score function $s(x, t) := \nabla_x \log p_t(x)$ and numerically integrating (7) from $x(T) \sim p_T$ down to $x(0)$.

Probability flow ODE (deterministic sampling). In addition to the reverse SDE, there exists a deterministic ODE whose solution has the same marginals $\{p_t\}$ as the SDE. This probability flow ODE is

$$dx = \left[f(x,t) - \tfrac{1}{2}g(t)^2\nabla_x \log p_t(x)\right] dt. \tag{8}$$

Thus, replacing $\nabla_x \log p_t(x)$ by a learned score model yields a deterministic sampler that often requires fewer steps but may reduce diversity compared to the stochastic reverse SDE [6].

Learning the score function and sampling. We parameterize a neural score model $s_\theta(x,t) \approx \nabla_x \log p_t(x)$ (implemented with the same U-Net as the DDPM baseline). Training uses denoising score matching (DSM) across continuous times. After training s_θ, we generate samples by either reverse SDE sampling, where we draw $x(T) \sim \mathcal{N}(0,I)$ and integrate (7) numerically (Euler–Maruyama), injecting Gaussian noise at each step, or probability flow ODE sampling, where we draw $x(T) \sim \mathcal{N}(0,I)$ and integrate (8) (Euler), yielding deterministic trajectories.

4.1.4 Direction 4: Classifier-Free Guidance for Conditional Diffusion

This section provides the theoretical background for Direction 4, where we extend the unconditional DDPM to class-conditional generation using classifier-free guidance (CFG) [1].

Conditional diffusion models. Let $c \in \mathcal{C}$ denote a class label (for MNIST, $\mathcal{C} = \{0,\dots,9\}$). A class-conditional diffusion model learns the conditional data distribution $p(x_0 \mid c)$ by defining a forward noising process

$$q(x_t \mid x_{t-1}) = \mathcal{N}\Big(x_t \mid \sqrt{1-\beta_t}\, x_{t-1},\ \beta_t I\Big),$$

which is independent of c, and a learned reverse process $p_\theta(x_{t-1} \mid x_t, c)$. As in standard DDPMs, this is parameterized via a conditional noise predictor $\epsilon_\theta(x_t,t,c)$ and trained using the denoising objective

$$\mathcal{L}_{\text{cond}}(\theta) = \mathbb{E}_{x_0,c,\epsilon,t}\left[\|\epsilon - \epsilon_\theta(x_t,t,c)\|_2^2\right],$$

where $x_t = \sqrt{\bar\alpha_t}x_0 + \sqrt{1-\bar\alpha_t}\,\epsilon$.

Joint conditional and unconditional training. Classifier-free guidance augments this setup by jointly training a conditional and an unconditional diffusion model using a single network. This is achieved by introducing a special null label $\emptyset$ and randomly dropping the conditioning during training:

$$c' = \begin{cases} c, & \text{with probability } 1 - p_{\text{uncond}}, \\ \emptyset, & \text{with probability } p_{\text{uncond}}. \end{cases}$$

The model is then trained with the same denoising objective

$$\mathcal{L}_{\text{CFG}}(\theta) = \mathbb{E}_{x_0,c,\epsilon,t}\left[\|\epsilon - \epsilon_\theta(x_t,t,c')\|_2^2\right],$$

so that the network learns to approximate both $\epsilon_\theta(x_t,t,c)$ (conditional) and $\epsilon_\theta(x_t,t,\emptyset)$ (unconditional) [1].

Guided sampling via score interpolation. At sampling time, classifier-free guidance modifies the reverse diffusion dynamics by combining conditional and unconditional predictions. Let $\epsilon_\theta(x_t,t,c)$ and $\epsilon_\theta(x_t,t,\emptyset)$ denote the conditional and unconditional noise estimates, respectively. The guided noise prediction is defined as

$$\tilde\epsilon_\theta(x_t,t,c) = (1+w)\,\epsilon_\theta(x_t,t,c) - w\,\epsilon_\theta(x_t,t,\emptyset), \tag{9}$$

where $w \geq 0$ is the guidance scale. This guided estimate replaces ϵ_θ in the DDPM reverse update, yielding a modified reverse process that biases sampling toward class-consistent regions of the data manifold.

Score-based interpretation. In the continuous-time limit, ϵ_θ is proportional to the score $\nabla_x \log p_t(x \mid c)$. Increasing the guidance scale w biases samples toward highly class-consistent modes, improving quality at the expense of diversity. While classifier guidance uses gradients of an auxiliary classifier $\nabla_x \log p(c \mid x_t)$, classifier-free guidance achieves a similar effect by combining conditional and unconditional score estimates within a single model.

4.1.5 Quantitative Metrics

To compare model variants with different objectives (DDPM ELBO, FM velocity regression, SDE score-matching, latent diffusion, and CFG), we use classifier-based metrics that are comparable across all approaches. We let $x \sim p_\theta$ be a generated image and let $p_\phi(y \mid x)$ be the class-probabilities from a separately trained MNIST classifier, and we let $f_\phi(x) \in \mathbb{R}^d$ denote the penultimate-layer feature embedding of the same classifier. We generate $N = 1000$ samples per model. For CFG we sample uniformly across classes (100 per class) at $w = 2.0$. All models are trained with the same horizon $T = 1000$, Adam (10^{-3}), batch size 256, and 50 epochs for this comparison (see Appendix 4.2.5 for code).

Inception Score (IS). We report the standard IS computed from classifier predictions:

$$\text{IS} = \exp\Big(\mathbb{E}_x\big[\text{KL}(p_\phi(y \mid x) \,\|\, p_\phi(y))\big]\Big), \qquad p_\phi(y) = \mathbb{E}_x[p_\phi(y \mid x)].$$

Higher IS means samples are confidently classifiable (quality) while the marginal class distribution is spread out (diversity).

Fréchet Inception Distance (FID). We compute FID between real and generated feature distributions using $f_\phi(x)$. Approximating each by a Gaussian with means/covariances (μ_r, Σ_r) and (μ_g, Σ_g),

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{tr}\Big(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\Big),$$

where lower is better and reflects closer alignment to the real data manifold in feature space.

Quality (classifier confidence). As a simple perceptual proxy, we report mean $\pm$ std of the max predicted probability:

$$\text{Qual} = \mathbb{E}_x\big[\max_y p_\phi(y \mid x)\big], \qquad \text{Std} = \text{Std}_x\big[\max_y p_\phi(y \mid x)\big].$$

Diversity (class entropy). We measure global class coverage as the entropy of the mean predicted class distribution:

$$\text{Div} = H\big(p_\phi(y)\big) = -\sum_{k=0}^{9} p_\phi(y{=}k)\log p_\phi(y{=}k),$$

with maximum $\log 10 \approx 2.30$ for a uniform class mix.

Final training loss. We also report the final training loss for each method, but emphasize it is not directly comparable across model families due to different objectives and parameterizations.

4.2 Code Changes for Directions in Part A

4.2.1 Direction 1: Code Changes

This direction converts the provided MNIST DDPM template into a Flow Matching (FM) baseline while keeping the `ScoreNet` U-Net, MNIST preprocessing, optimizer, and scheduler unchanged. The key change is that we replace the diffusion Markov chain training/sampling logic with a continuous-time ODE model that learns a velocity field.

High-level mapping (DDPM $\rightarrow$ FM). In the DDPM template, the wrapper class `DDPM` provides (i) a discrete-time forward noising process $q(x_t \mid x_0)$, (ii) a learned reverse process $p_\theta(x_{t-1} \mid x_t)$ used for sampling, and (iii) the simplified ELBO implemented as an MSE loss on the noise prediction $\epsilon_\theta(x_t, t)$. In the FM conversion:

- We keep the same `ScoreNet` and MNIST preprocessing, but replace the discrete diffusion schedule (`beta/alpha/alpha_bar`) and the functions `forward_diffusion` and `reverse_diffusion` with a continuous-time path and velocity regression objective.
- We sample continuous time $t \sim \mathcal{U}(0,1)$ and construct the conditional OT / linear Gaussian path via
$$X_t = (1-t)X_0 + tX_1, \quad X_0 \sim \mathcal{N}(0, I), \;\; X_1 \sim q.$$
- We train a neural velocity field $u_\theta(X_t, t)$ by regressing to the closed-form target velocity $X_1 - X_0$ using MSE.
- We sample by integrating the ODE $dX/dt = u_\theta(X, t)$ from $t = 0$ to $t = 1$ using an Euler solver with a configurable number of steps.

(1) Network interface: keep `ScoreNet`, reinterpret output as velocity. The DDPM wrapper flattens MNIST and normalizes the discrete time index by dividing by T before passing it to `ScoreNet`:

```
self._network = network
self.network = lambda x, t: (
    self._network(x.reshape(-1, 1, 28, 28), (t.squeeze() / T))
).reshape(-1, 28 * 28)
```

For Flow Matching, we keep the same backbone but pass continuous $t \in [0, 1]$ directly, and interpret the output as a velocity in data space. The resulting model includes a `velocity` helper that mirrors the flatten convention:

```
class FlowMatchingModel(nn.Module):
    def __init__(self, network):
        super().__init__()
        self.network = network

    def velocity(self, x_flat, t):
        if t.ndim == 2:
            t_in = t.squeeze(1)
        else:
            t_in = t
        v = self.network(x_flat.reshape(-1, 1, 28, 28), t_in).reshape(-1, 28*28)
        return v
```

(2) Training objective: replace DDPM `elbo_simple` with FM regression loss. In DDPM, `elbo_simple` samples a discrete timestep, forms x_t by forward diffusion, and minimizes MSE between sampled noise and predicted noise:

```
def elbo_simple(self, x0):
    t = torch.randint(1, self.T, (x0.shape[0], 1)).to(x0.device)

    epsilon = torch.randn_like(x0)
    xt = self.forward_diffusion(x0, t, epsilon)

    return -nn.MSELoss(reduction="mean")(epsilon, self.network(xt, t))

def loss(self, x0):
    return -self.elbo_simple(x0).mean()
```

For FM, we no longer compute x_t using `alpha_bar` and ϵ. Instead, each minibatch draws $X_0 \sim \mathcal{N}(0, I)$, samples $t \sim \mathcal{U}(0, 1)$, constructs $X_t = (1-t)X_0 + tX_1$, and regresses $u_\theta(X_t, t)$ to $X_1 - X_0$:

```
    def loss(self, x1_flat):
        B = x1_flat.shape[0]
        device = x1_flat.device

        x0 = torch.randn_like(x1_flat)
        t = torch.rand(B, 1, device=device)
        xt = (1.0 - t) * x0 + t * x1_flat
        v_target = x1_flat - x0
        v_pred = self.velocity(xt, t)

        return F.mse_loss(v_pred, v_target, reduction="mean")
```

(3) Sampling: replace stochastic reverse diffusion with ODE integration. DDPM sampling starts from $x_T \sim \mathcal{N}(0, I)$ and iterates the learned reverse transitions while injecting noise:

```
@torch.no_grad()
def sample(self, shape):
    xT = torch.randn(shape).to(self.beta.device)
    xt = xT
    for t in range(self.T, 0, -1):
        noise = torch.randn_like(xT) if t > 1 else 0
        t = torch.tensor(t).expand(xt.shape[0], 1).to(self.beta.device)
        xt = self.reverse_diffusion(xt, t, noise)
    return xt
```

In FM, sampling is deterministic, so we draw $X(0) \sim \mathcal{N}(0, I)$ and integrate $dX/dt = u_\theta(X, t)$ with Euler steps:

```
    @torch.no_grad()
    def sample(self, nsamples, n_steps=50):
        device = next(self.parameters()).device
        x = torch.randn(nsamples, 28*28, device=device)

        t_grid = torch.linspace(0.0, 1.0, n_steps + 1, device=device)
        for k in range(n_steps):
            t_k = t_grid[k].expand(nsamples, 1)
            dt = (t_grid[k+1] - t_grid[k]).item()
            x = x + dt * self.velocity(x, t_k)

        return x
```

(4) Training loop: minimal change. We keep the same optimizer and scheduler type as in the DDPM template (`Adam` and `ExponentialLR`). The change is that the FM training loop calls `model.loss(x1)` directly (no dependence on T):

```
fm_unet = ScoreNet((lambda t: torch.ones(1).to(device))).to(device)
fm_model = FlowMatchingModel(fm_unet).to(device)

fm_optimizer = torch.optim.Adam(fm_model.parameters(), lr=learning_rate)
fm_scheduler = torch.optim.lr_scheduler.ExponentialLR(fm_optimizer, 0.9999)

def train_flow_matching(model, optimizer, scheduler, dataloader, epochs, device,
                        per_epoch_callback=None):
    total_steps = len(dataloader) * epochs
    progress_bar = tqdm(range(total_steps), desc="Training (Flow Matching)")

    loss_history = []

    for epoch in range(epochs):
        model.train()
        for x1, _ in dataloader:
            x1 = x1.to(device)
            optimizer.zero_grad()
            loss = model.loss(x1)
            loss.backward()
            optimizer.step()
            scheduler.step()

            loss_history.append(loss.item())

    return loss_history
```

Overall, the conversion required no changes to `ScoreNet` or the data pipeline. The main edits were (i) replacing the diffusion schedule and the noise-prediction ELBO with a continuous-time conditional path and velocity regression loss, and (ii) replacing ancestral sampling with deterministic ODE integration.

4.2.2 Direction 2: Code Changes

Here, we implement a latent diffusion baseline by (i) pretraining an autoencoder E, D on MNIST, (ii) freezing it, and (iii) training a DDPM in the latent space $z = E(x)$ instead of pixel space. We keep the same data pipeline and noise schedule, but add an autoencoder stage and replace the pixel-space `ScoreNet` with a smaller network on d-dimensional latents.

High-level mapping (pixel DDPM $\rightarrow$ latent DDPM). Starting from the original template `DDPM` and `train` loop:

- We add an autoencoder (`Encoder`, `Decoder`, `Autoencoder`) trained with an MSE reconstruction loss.
- We freeze the trained autoencoder and define a latent-space DDPM wrapper `LatentDDPM` that runs the same forward/reverse diffusion logic on $z \in \mathbb{R}^d$.
- We introduce a smaller latent network `LatentUNet` (MLP with time embeddings) to predict noise in latent space.
- Sampling is performed by reverse diffusion in latent space, followed by decoding with `autoencoder.decode`.

(1) Add an autoencoder and training loop. We introduce a convolutional encoder/decoder pair and wrap them in `Autoencoder`. The reconstruction objective is implemented directly as:

```python
class Autoencoder(nn.Module):
    def __init__(self, latent_dim=32):
        super().__init__()
        self.latent_dim = latent_dim
        self.encoder = Encoder(latent_dim)
        self.decoder = Decoder(latent_dim)

    def encode(self, x):
        if x.ndim == 2:
            x = x.reshape(-1, 1, 28, 28)
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

    def loss(self, x):
        if x.ndim == 2:
            x = x.reshape(-1, 1, 28, 28)
        x_recon, _ = self.forward(x)
        return F.mse_loss(x_recon, x, reduction="mean")
```

and the corresponding training loop:

```python
def train_autoencoder(model, optimizer, scheduler, dataloader, epochs, device,
                      per_epoch_callback=None):
    total_steps = len(dataloader) * epochs
    progress_bar = tqdm(range(total_steps), desc="Training Autoencoder")

    loss_history = []

    for epoch in range(epochs):
        model.train()
        for x, _ in dataloader:
            x = x.to(device)
            if x.ndim == 2:
                x = x.reshape(-1, 1, 28, 28)

            optimizer.zero_grad()
            loss = model.loss(x)
            loss.backward()
            optimizer.step()
            scheduler.step()

            loss_history.append(loss.item())

    return loss_history
```

(2) Replace the pixel U-Net with a latent network. Since the latent dimension is $d = 32$, we replace the image U-Net `ScoreNet` with a smaller latent network that takes (z, t) and predicts noise:

```
class LatentUNet(nn.Module):
    def __init__(self, latent_dim=32, hidden_dim=256, embed_dim=128):
        super().__init__()
        self.latent_dim = latent_dim

        self.time_embed = nn.Sequential(
            GaussianFourierProjection(embed_dim=embed_dim),
            nn.Linear(embed_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
        )
        ...
    def forward(self, z, t):
        t_emb = self.time_embed(t)
        ...
        return self.output_proj(h)
```

(3) Implement latent DDPM by reusing the same schedule. We wrap the autoencoder and latent network in `LatentDDPM`. The autoencoder is frozen, the same $\beta/\alpha/\bar{\alpha}$ buffers are registered, and diffusion is performed in latent space:

```
class LatentDDPM(nn.Module):
    def __init__(self, autoencoder, network, T=1000, beta_1=1e-4, beta_T=2e-2):
        super().__init__()

        self.autoencoder = autoencoder
        for param in self.autoencoder.parameters():
            param.requires_grad = False

        self.network = network
        self.T = T

        self.register_buffer("beta", torch.linspace(beta_1, beta_T, T + 1))
        self.register_buffer("alpha", 1 - self.beta)
        self.register_buffer("alpha_bar", self.alpha.cumprod(dim=0))
```

Training encodes $x \mapsto z_0$ once per minibatch (under `torch.no_grad()`), then applies the same MSE loss as DDPM:

```
    def loss(self, x):
        self.autoencoder.eval()
        with torch.no_grad():
            z0 = self.autoencoder.encode(x.reshape(-1, 1, 28, 28))

        t = torch.randint(1, self.T, (z0.shape[0], 1), device=x.device)
        epsilon = torch.randn_like(z0)
        zt = self.forward_diffusion(z0, t, epsilon)

        t_normalized = t.squeeze().float() / self.T
        epsilon_pred = self.network(zt, t_normalized)

        return F.mse_loss(epsilon_pred, epsilon, reduction="mean")
```

(4) Sampling: reverse diffusion in latent space, then decode. Sampling mirrors the DDPM template but operates on z_t and decodes at the end:

```
    @torch.no_grad()
    def sample(self, nsamples):
        device = self.beta.device
        latent_dim = self.autoencoder.latent_dim

        zt = torch.randn(nsamples, latent_dim, device=device)

        for t in range(self.T, 0, -1):
            noise = torch.randn_like(zt) if t > 1 else torch.zeros_like(zt)
            t_tensor = torch.tensor(t, device=device).expand(nsamples, 1)
            zt = self.reverse_diffusion(zt, t_tensor, noise)

        self.autoencoder.eval()
        x_decoded = self.autoencoder.decode(zt)

        return x_decoded.reshape(nsamples, -1)
```

Overall, the changes are that we add an autoencoder stage, swap the pixel network for a latent network, and reuse the DDPM schedule and loss in latent space while decoding generated latents back to pixels.

4.2.3 Direction 3: Code Changes

This direction replaces the discrete-time DDPM training/sampling logic with a continuous-time variance-preserving SDE (VP-SDE) formulation, while keeping the same `ScoreNet` U-Net. Concretely, the main changes are (i) switching from a discrete noise schedule $\{\beta_t\}_{t=1}^T$ to a continuous schedule $\beta(t)$ on $t \in (0, 1]$, (ii) training the network as a score model via continuous-time denoising score matching rather than discrete noise-prediction, and (iii) sampling by numerically integrating either the reverse-time SDE (stochastic) or the probability flow ODE (deterministic).

High-level mapping. Starting from the DDPM template, which predicts injected noise ϵ at an integer timestep and trains with an MSE loss, the VP-SDE implementation makes three conceptual edits:

- **Continuous time.** Replace integer timesteps $t \in \{1, \ldots, T\}$ with continuous $t \in (0, 1]$ sampled from a uniform distribution. The noise level is controlled by a continuous function $\beta(t)$ (linear between `beta_min` and `beta_max`).
- **Closed-form perturbation kernel.** Replace `DDPM.forward_diffusion` by the VP-SDE marginal.
- **Score training and SDE/ODE sampling.** Interpret the U-Net output as a noise estimate and convert it to a score via $s_\theta(x, t) = -\epsilon_\theta(x, t)/\sigma(t)$. Train using continuous-time denoising score matching, and sample by integrating either the reverse SDE (Euler–Maruyama) or the probability flow ODE (Euler).

(1) VP-SDE schedule and closed-form marginals. Instead of storing discrete `beta/alpha/alpha_bar`, the VP-SDE wrapper defines a continuous noise schedule $\beta(t)$ and uses its integral to compute the perturbation:

```
class VPSDE(nn.Module):
    def __init__(self, network, beta_min=0.1, beta_max=20.0, T=1.0):
        super().__init__()
        self._network = network
        self.beta_min, self.beta_max, self.T = beta_min, beta_max, T

    def beta(self, t):
        return self.beta_min + t * (self.beta_max - self.beta_min)

    def integral_beta(self, t):
        return self.beta_min * t + 0.5 * t**2 * (self.beta_max - self.beta_min)

    def marginal_prob_mean_coeff(self, t):
        return torch.exp(-0.5 * self.integral_beta(t))

    def marginal_prob_std(self, t):
        return torch.sqrt(1.0 - torch.exp(-self.integral_beta(t)))

    def forward_sample(self, x0, t, epsilon):
        a = self.marginal_prob_mean_coeff(t)[:, None]
        s = self.marginal_prob_std(t)[:, None]
        return a * x0 + s * epsilon
```

(2) Noise prediction $\rightarrow$ score prediction (normalization by $\sigma(t)$). The original DDPM template predicts injected noise ϵ directly. In the VP-SDE code, we still run the same U-Net, but reinterpret its output as a noise estimate $\epsilon_\theta(x, t)$ and convert it into a score estimate $s_\theta(x, t) = -\epsilon_\theta(x, t)/\sigma(t)$:

```
    def score(self, x, t):
        if t.ndim == 2:
            t = t.squeeze(1)

        t_normalized = t / self.T
        x_img = x.reshape(-1, 1, 28, 28)

        output = self._network(x_img, t_normalized).reshape(-1, 28 * 28)

        std = self.marginal_prob_std(t)[:, None]
        return -output / std
```

(3) Continuous-time denoising score matching loss. Training switches from the discrete DDPM objective $\|\epsilon - \epsilon_\theta(x_t, t)\|^2$ to a continuous-time denoising score matching objective. We sample $t \sim \mathcal{U}(0, 1]$, generate $x(t) = \alpha(t)x(0) + \sigma(t)\epsilon$, and regress the predicted score to the analytic target score $s^\star(x(t), t) = -\epsilon/\sigma(t)$:

```
    def loss(self, x0):
        batch_size = x0.shape[0]
        device = x0.device

        eps = 1e-5
        t = torch.rand(batch_size, device=device) * (self.T - eps) + eps
        epsilon = torch.randn_like(x0)

        xt = self.forward_sample(x0, t, epsilon)
        score_pred = self.score(xt, t)
        std = self.marginal_prob_std(t)[:, None]
        score_true = -epsilon / std

        loss = torch.mean(torch.sum((score_pred - score_true)**2 * std**2, dim=1))
        return loss
```

(4) Sampling: reverse SDE (Euler–Maruyama) vs. probability flow ODE (Euler). DDPM sampling iterates a discrete reverse Markov chain. The VP-SDE implementation instead integrates dynamics backward in continuous time. We provide both a stochastic sampler (reverse SDE) and a deterministic sampler (probability flow ODE):

```
    @torch.no_grad()
    def sample_sde(self, shape, n_steps=1000):
        device = next(self.parameters()).device
        x = torch.randn(shape, device=device)
        dt = -self.T / n_steps

        for i in range(n_steps):
            t = self.T - i * self.T / n_steps
            t_tensor = torch.full((shape[0],), t, device=device)

            beta_t = self.beta(t)
            f = -0.5 * beta_t * x
            g = torch.sqrt(torch.tensor(beta_t, device=device))
            score = self.score(x, t_tensor)
            drift = f - g**2 * score

            noise = torch.randn_like(x) if i < n_steps - 1 else 0
            x = x + drift * dt + g * torch.sqrt(torch.abs(torch.tensor(dt))) * noise
        return x

    @torch.no_grad()
    def sample_ode(self, shape, n_steps=1000):
        device = next(self.parameters()).device
        x = torch.randn(shape, device=device)
        dt = -self.T / n_steps

        for i in range(n_steps):
            t = self.T - i * self.T / n_steps
            t_tensor = torch.full((shape[0],), t, device=device)

            beta_t = self.beta(t)
            f = -0.5 * beta_t * x
            g_squared = beta_t

            score = self.score(x, t_tensor)
            drift = f - 0.5 * g_squared * score
            x = x + drift * dt
        return x
```

4.2.4 Direction 4: Code Changes

This direction implements classifier-free guidance (CFG) by extending the MNIST DDPM into a class-conditional DDPM trained with label dropout. The core code changes are: (i) augment `ScoreNet` to accept class labels via a learned embedding, (ii) modify the DDPM wrapper so the network takes (x_t, t, y), (iii) change the loss to randomly drop labels with probability p_{uncond}, and (iv) change sampling to combine conditional and unconditional predictions with guidance scale w.

High-level mapping (unconditional DDPM $\to$ CFG DDPM). In the original template, `ScoreNet.forward` takes only (x, t) and the wrapper `DDPM` learns a noise predictor $\epsilon_\theta(x_t, t)$, trained with the simplified ELBO MSE loss and sampled via the standard reverse chain. In CFG, we

- Replace `ScoreNet` with `ConditionalScoreNet` that takes an additional label input y and forms a joint embedding by concatenating time and class embeddings.
- Replace `DDPM` with `ClassifierFreeDDPM`, which (a) supports a dedicated null label, (b) drops labels during training with probability `p_uncond`, and (c) performs guided sampling by mixing conditional and unconditional noise predictions.
- Replace `train` with `train_cfg`, since the model loss now requires labels (x, y) rather than only x.

(1) Network interface: add class conditioning to `ScoreNet`. The original `ScoreNet` conditions only on time via Gaussian Fourier features:

```
def forward(self, x, t):
    embed = self.act(self.embed(t))
    ...
```

Direction 4 introduces `ConditionalScoreNet`, adding a class embedding table and concatenating class/time embeddings before injecting them through the same `Dense` blocks:

```
self.time_embed = nn.Sequential(
    GaussianFourierProjection(embed_dim=embed_dim),
    nn.Linear(embed_dim, embed_dim),
)

self.class_embed = nn.Embedding(num_classes + 1, embed_dim)

self.dense1 = Dense(embed_dim * 2, channels[0])
...
```

and the forward signature becomes (x, t, y):

```
def forward(self, x, t, y):
    t_embed = self.act(self.time_embed(t))
    y_embed = self.class_embed(y)

    embed = torch.cat([t_embed, y_embed], dim=-1)
    ...
```

(2) DDPM wrapper: pass labels through the network. The original wrapper normalizes time and calls `ScoreNet(x,t)`:

```
self.network = lambda x, t: (
    self._network(x.reshape(-1, 1, 28, 28), (t.squeeze() / T))
).reshape(-1, 28 * 28)
```

Direction 4 replaces this with a conditional call that threads labels into the backbone:

```
self.network = lambda x, t, y: (
    self._network(x.reshape(-1, 1, 28, 28), (t.squeeze() / T), y)
).reshape(-1, 28 * 28)
```

It also defines a dedicated null label index (the extra entry in `nn.Embedding(num_classes + 1, ...)`) used for unconditional training and sampling:

```
self.num_classes = num_classes
self.null_class = num_classes
self.p_uncond = p_uncond
self.guidance_scale = guidance_scale
```

(3) Training objective: label dropout for classifier-free guidance. The original DDPM loss predicts ϵ at discrete t:

```
t = torch.randint(1, self.T, (x0.shape[0], 1)).to(x0.device)
epsilon = torch.randn_like(x0)
xt = self.forward_diffusion(x0, t, epsilon)
return -nn.MSELoss(reduction="mean")(epsilon, self.network(xt, t))
```

Direction 4 changes the loss signature to `loss(self, x0, y)` and randomly drops conditioning labels with probability `p_uncond` by replacing them with `null_class`:

```
mask = torch.rand(batch_size, device=device) < self.p_uncond
y_train = y.clone()
y_train[mask] = self.null_class
```

```
epsilon_pred = self.network(xt, t, y_train)

return F.mse_loss(epsilon_pred, epsilon, reduction="mean")
```

This trains a single model to operate both conditionally ($y \in \{0, \dots, 9\}$) and unconditionally ($y = \emptyset$ implemented as `null_class`).

(4) Sampling: guided noise prediction ϵ_{guided}. The original reverse step uses a single noise prediction `self.network(xt,t)`. Direction 4 computes both conditional and unconditional predictions and combines them using the CFG scale `guidance_scale`:

```
eps_cond = self.network(xt, t, y)

y_uncond = torch.full_like(y, self.null_class)
eps_uncond = self.network(xt, t, y_uncond)

eps_guided = eps_uncond + self.guidance_scale * (eps_cond - eps_uncond)
```

The remainder of the reverse diffusion step is unchanged except that it substitutes `eps_guided` for the noise prediction:

```
mean = (
    1.0 / torch.sqrt(self.alpha[t]) * (
        xt - (self.beta[t]) / torch.sqrt(1 - self.alpha_bar[t]) * eps_guided
    )
)
```

Sampling now supports explicit labels and an optional override of `guidance_scale`:

```
@torch.no_grad()
def sample(self, shape, y=None, guidance_scale=None):
    ...
```

(5) Training loop: `train` $\rightarrow$ `train_cfg`. Because the loss requires labels, Direction 4 replaces the original loop that iterates `for x, _ in dataloader` with a loop that consumes `(x,y)` and calls `model.loss(x,y)`:

```
for x, y in dataloader:
    x = x.to(device)
    y = y.to(device)

    optimizer.zero_grad()
    loss = model.loss(x, y)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

This is the only required change to the training-loop. The optimizer and scheduler usage are identical to the baseline DDPM.

4.2.5 Code for Quantitative Metrics

All metrics in Table 1 are computed using predictions and penultimate-layer features from a lightweight CNN trained on MNIST. The classifier is trained once (5 epochs, Adam, `lr=1e-3`) and then frozen. For each generative model, we draw $N = 1000$ samples (normalized to match the classifier preprocessing) and compute (i) Inception Score (IS) from classifier probabilities, (ii) FID from Gaussian statistics of classifier feature embeddings, (iii) sample quality as the mean $\pm$ std of the maximum predicted class probability, and (iv) diversity as the entropy of the mean predicted class distribution. For FID, we compute real-data features from a single batch of 1000 MNIST training images and compare against generated features.

Classifier and feature extractor. We use a two-convolution CNN and take the activations before the final linear layer as features:

```
class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x)); x = self.pool(x)
        x = F.relu(self.conv2(x)); x = self.pool(x)
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        return self.fc2(x)

    def get_features(self, x):
        x = F.relu(self.conv1(x)); x = self.pool(x)
        x = F.relu(self.conv2(x)); x = self.pool(x)
        x = x.view(-1, 64 * 7 * 7)
        return F.relu(self.fc1(x))
```

Metrics. IS is computed by splitting generated samples into 10 chunks and averaging $\exp(\mathbb{E}[\mathrm{KL}(p(y \mid x) \| p(y))])$ across splits. FID is computed from feature means/covariances using a matrix square root. Quality is the mean max softmax probability, and diversity is the entropy of the mean softmax vector:

```
def calculate_inception_score(samples, classifier, splits=10):
    classifier.eval()
    with torch.no_grad():
        preds = F.softmax(classifier(samples), dim=1).cpu().numpy()

        split_scores = []
        for k in range(splits):
            part = preds[k * (len(preds) // splits): (k + 1) * (len(preds) // splits), :]
            py = np.mean(part, axis=0)
            scores = []
            for i in range(part.shape[0]):
                pyx = part[i, :]
                scores.append(entropy(pyx, py))
            split_scores.append(np.exp(np.mean(scores)))

        return np.mean(split_scores), np.std(split_scores)

def calculate_fid(real_features, fake_features):
    mu1, sigma1 = real_features.mean(axis=0), np.cov(real_features, rowvar=False)
    mu2, sigma2 = fake_features.mean(axis=0), np.cov(fake_features, rowvar=False)

    ssdiff = np.sum((mu1 - mu2) ** 2)
    covmean = sqrtm(sigma1.dot(sigma2))

    fid = ssdiff + np.trace(sigma1 + sigma2 - 2 * covmean)
    return fid
```

```
def calculate_quality_score(samples, classifier):
    classifier.eval()
    with torch.no_grad():
        preds = F.softmax(classifier(samples), dim=1)
        max_probs = preds.max(dim=1)[0]
        return max_probs.mean().item(), max_probs.std().item()

def calculate_diversity_score(samples, classifier):
    classifier.eval()
    with torch.no_grad():
        preds = F.softmax(classifier(samples), dim=1)
        avg_pred = preds.mean(dim=0).cpu().numpy()
        return entropy(avg_pred)
```

Evaluation pipeline. We first train the MNIST classifier, cache real features from 1000 training images, and then evaluate each generative model by extracting features and computing all metrics on $N = 1000$ generated samples. This keeps the evaluation consistent across the different model directions while relying on the same feature space and preprocessing.

4.3 Failure Modes for Alternative Directions in Part A

4.3.1 Direction 1: Failure Modes

Here, we summarize three failure modes for Flow Matching (Direction 1): (i) too few ODE integration steps at sampling time, (ii) inadequate training (early stopping), and (iii) optimizer instability at overly high learning rates. In all experiments, we fix the random seed (`torch.manual_seed(42)`) to make comparisons consistent across settings.

(1) Too few ODE steps. Flow Matching generates samples by numerically integrating an ODE from $t = 0$ to $t = 1$. Figure 8 shows that using very few Euler steps (e.g., 1 or 5) yields blurry or incomplete digits, since discretization error accumulates and trajectories deviate from the data manifold. As the number of steps increases (10–20), digit structure becomes clearer, and at 200–1000 steps samples are sharper and more stable.

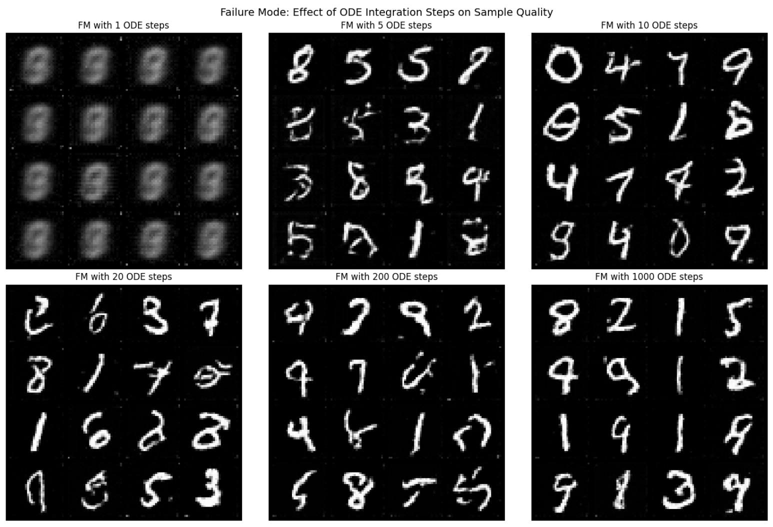


Figure 8: Effect of the number of ODE integration steps on Flow Matching samples (Euler solver).

(2) Inadequate training (early stopping). Even with a fixed sampling budget (50 Euler steps), undertrained models produce noisy and inconsistent generations. Figure 9 compares models trained for 1, 5, 10, 20, 50, and 100 epochs. With very few epochs, the velocity field is poorly learned and samples contain fragmented strokes and background noise. Sample quality improves steadily with training duration, with the clearest digits appearing after longer training.

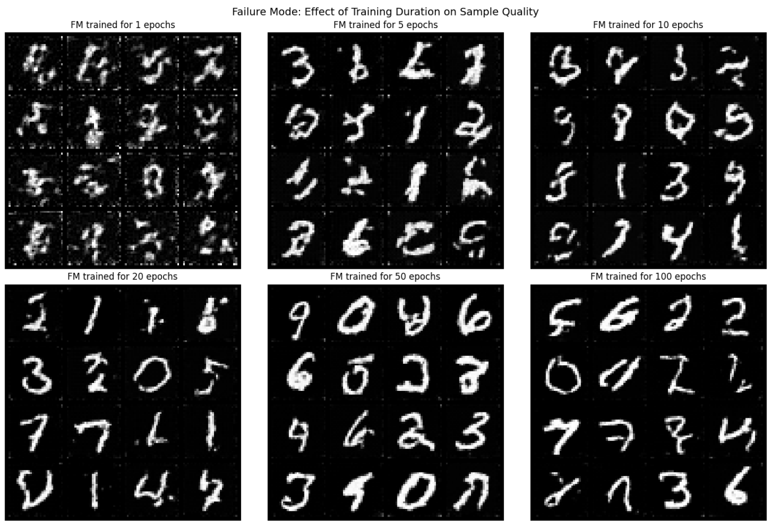


Figure 9: Effect of training duration (epochs) on Flow Matching sample quality (sampling with 50 ODE steps).

(3) Learning-rate instability. Finally, we examine the effect of the optimizer learning rate on Flow Matching performance. Figure 10 shows samples obtained after training with different learning rates, while keeping all other hyperparameters fixed. Very small learning rates (e.g., 10^{-5}) lead to slow convergence and severely underfit, noisy samples. Larger learning rates produce clearer digits, with intermediate values (10^{-3}–10^{-4}) yielding the most stable training and best sample quality.

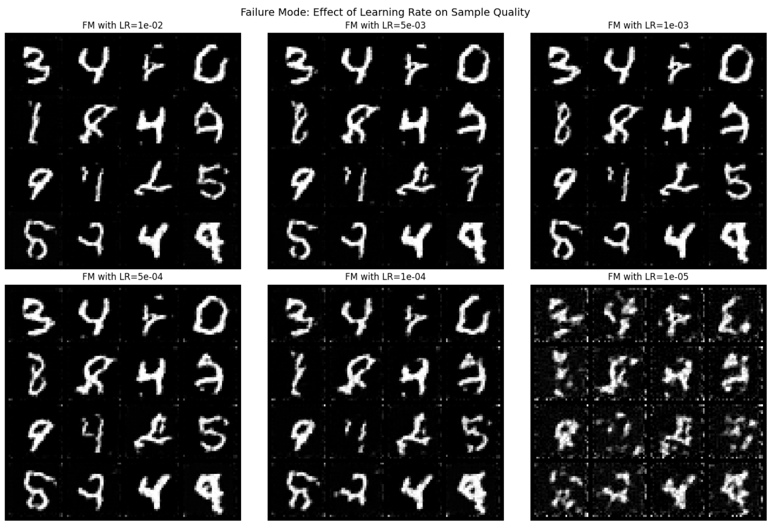


Figure 10: Effect of learning rate on Flow Matching sample quality (50 ODE steps, 30 training epochs).

4.3.2 Direction 2: Failure Modes

Here, we summarize two failure modes for latent diffusion (Direction 2): (i) using too few reverse-diffusion steps at sampling time, and (ii) choosing an incorrect initial noise scale for sampling in latent space. As in Direction 1, we fix the random seed (`torch.manual_seed(42)`) to make comparisons consistent across settings.

(1) Too few diffusion steps during sampling. The latent DDPM generates samples by running the reverse Markov chain from z_T to z_0. Figure 11 shows that using a heavily subsampled reverse chain (e.g., 10–100 steps) yields extremely incomplete digits, since large effective step sizes introduce discretization error and prevent the chain from denoising properly. As the number of steps increases (500), digit structure becomes more visible but remains noisy, while using the full chain (1000 steps) produces the clearest and most stable samples.

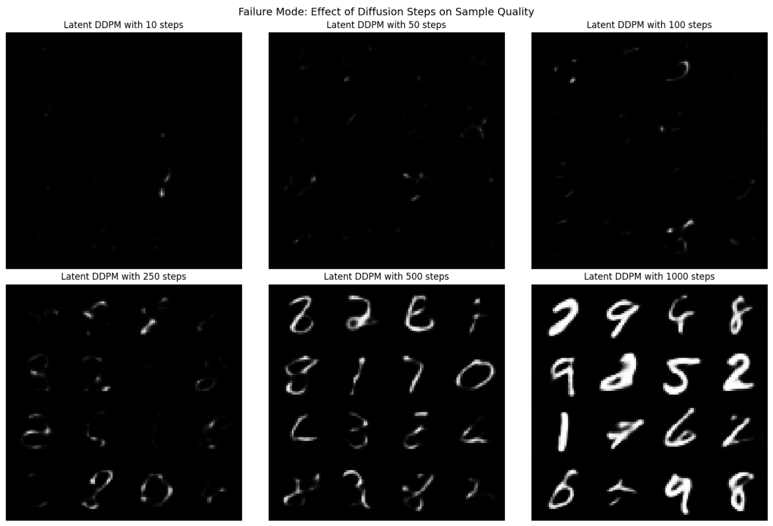

Failure Mode: Effect of Diffusion Steps on Sample Quality
Latent DDPM with 10 steps
Latent DDPM with 50 steps
Latent DDPM with 100 steps
Latent DDPM with 250 steps
Latent DDPM with 500 steps
Latent DDPM with 1000 steps

Figure 11: Effect of the number of reverse-diffusion steps on latent DDPM sample quality (subsampled reverse chain).

(2) Wrong initial noise scale in latent space. In the DDPM template, sampling starts from $z_T \sim \mathcal{N}(0, I)$. In latent diffusion this can be sensitive to the scaling of the learned latent space. If the initial noise magnitude is mismatched to what the latent DDPM was trained on, the reverse process can drift into regions that decode poorly. Figure 12 shows that small-to-moderate scales (0.1–1.0) still yield readable digits, whereas larger scales already degrade quality (2.0 introduces strong artifacts and partial collapse), and very large scales (5.0–10.0) break decoding entirely. This indicates that latent-space normalization and the assumed prior scale have a direct impact on sampling stability.

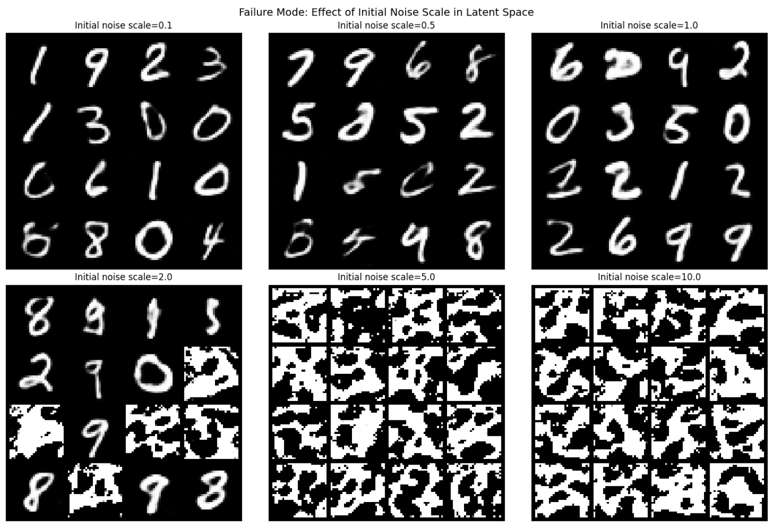


Figure 12: Effect of the initial latent noise scale on latent DDPM sample quality (full 1000-step reverse diffusion).

4.3.3 Direction 3: Failure Modes

This appendix summarizes two failure modes for the continuous-time VP-SDE model (Direction 3): (i) numerical instability when the variance schedule $\beta(t)$ grows too aggressively, and (ii) sensitivity to miscalibrated initial noise when sampling with the probability flow ODE. We fix the random seed (`torch.manual_seed(42)`) to make comparisons consistent.

(1) Numerical instability for rapidly growing $\beta(t)$. VP-SDE sampling depends on the choice of $\beta(t)$, since it controls both the drift and diffusion magnitude across time. Figure 13 varies the growth rate by increasing $\beta_{\max}$ while keeping $\beta_{\min} = 0.1$ fixed. With a moderate schedule ($\beta_{\max} = 20$) the ODE sampler produces recognizable digits, while too small $\beta_{\max}$ (e.g., 5) yields under-denoised, noisy samples. As $\beta_{\max}$ increases (50–200), the sampler becomes increasingly unstable and produces speckled outputs. At extreme values (500) the reverse-time integration collapses entirely.

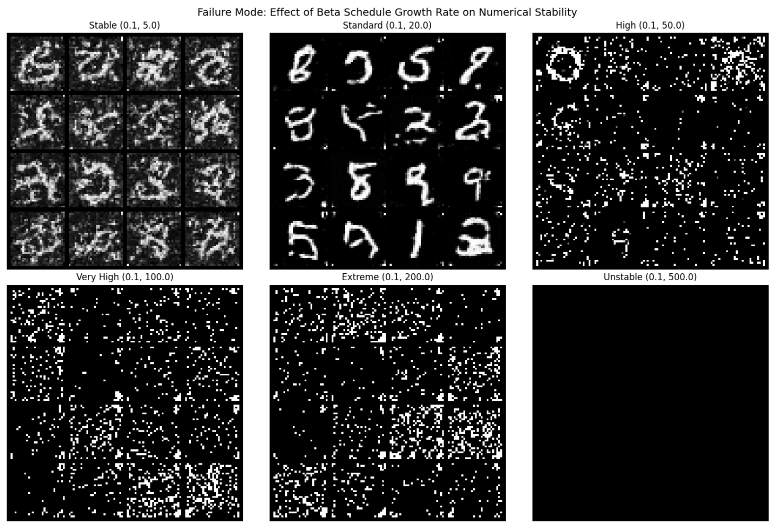


Figure 13: Effect of $\beta(t)$ growth rate on VP-SDE numerical stability (probability flow ODE sampling, 500 steps).

(2) Miscalibrated initial noise in ODE sampling. Probability flow ODE sampling is deterministic and starts from an initial Gaussian state that should match the model's assumed terminal distribution. Figure 14 shows that changing the initial noise scale can strongly affect sample quality, where small scales (0.1) lead to overly smooth, low-diversity outputs that can collapse to simple strokes, while very large scales (2.0 and above) yield noisy, unstable generations as trajectories begin too far from the typical set. The default scale (1.0) is the most reliable in our experiments, indicating that the initial distribution must be calibrated to the VP-SDE parameterization to avoid over-smoothing or noisy outputs.

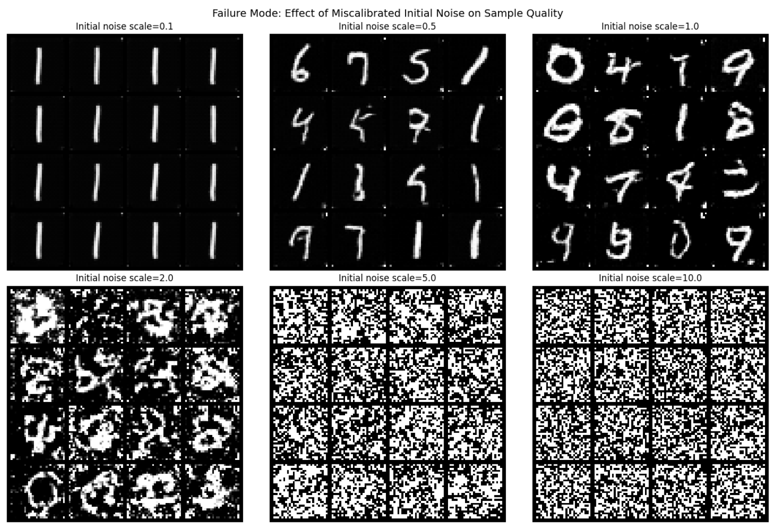


Figure 14: Effect of initial noise scale on VP-SDE samples (probability flow ODE sampling, 500 steps).

4.3.4 Direction 4: Failure Modes

This appendix summarizes two failure modes for classifier-free guidance (CFG): (i) overly large guidance scales, which reduce diversity and cause mode collapse, and (ii) insufficient unconditional training, which breaks the guidance mechanism. As in the other directions, we fix the random seed (`torch.manual_seed(42)`) to ensure consistent comparisons.

(1) Effect of guidance scale on sample quality and diversity. Classifier-free guidance interpolates between unconditional and conditional predictions using a guidance scale w. Figure 15 shows samples generated for a fixed digit class while varying w. With $w = 0$, the model reduces to unconditional sampling and produces diverse but weakly class-aligned digits. Moderate values ($w = 0.5$–2.0) yield sharper, more class-consistent samples. However, as w increases further ($w \geq 5$), diversity rapidly collapses and the model produces near-identical digits. At very large scales ($w = 10$), overconfident shapes appear, indicating that excessive guidance amplifies errors in the conditional score estimate.

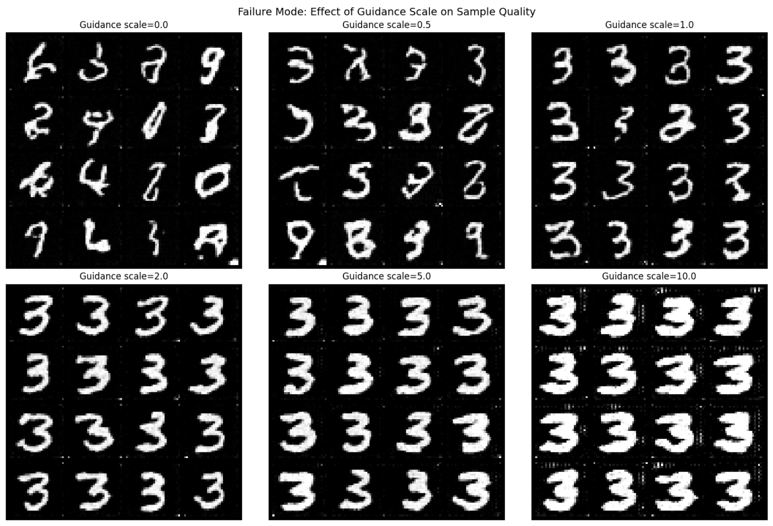


Figure 15: Effect of guidance scale w on classifier-free guidance samples for a fixed digit class.

(2) Insufficient unconditional training (p_{uncond} too low or high). CFG relies on the model learning conditional and unconditional score estimates during training. Figure 16 illustrates the effect of varying the unconditional training probability p_{uncond}. When $p_{\text{uncond}} = 0$, the model never observes unconditional inputs, causing CFG sampling to fail entirely. Small values ($p_{\text{uncond}} = 0.05$–0.2) yield stable training and high-quality guided samples. Large values ($p_{\text{uncond}} = 0.5$ or 1.0) bias the model toward unconditional behavior, degrading class specificity and producing noisy or unstable outputs.

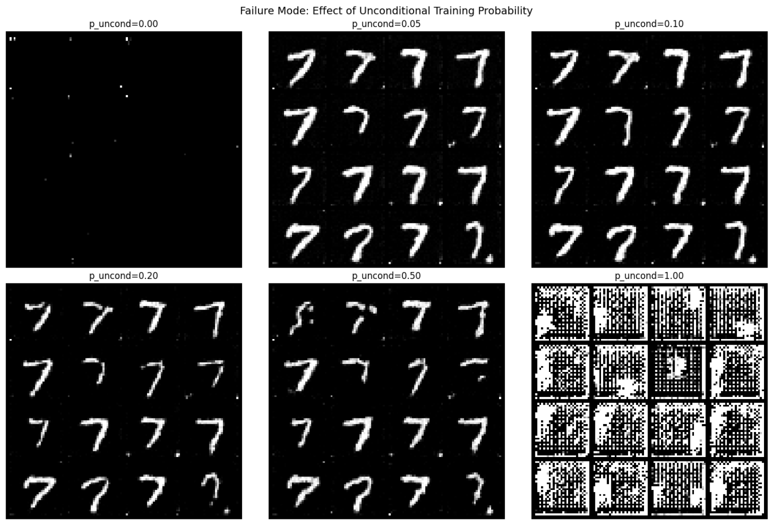


Figure 16: Effect of the unconditional training probability p_{uncond} on CFG sample quality.

5 Appendix for Part B

5.1 Marginal Likelihood Optimization Sanity Check

To assess the robustness of the marginal log-likelihood (MLL) optimization and mitigate sensitivity to local optima, we performed a coarse grid search over the kernel hyperparameters and compared the resulting MLL values with those obtained using local L-BFGS-B optimization.

For each model configuration, the grid search was used to identify a high-likelihood region in hyperparameter space. The L-BFGS-B optimizer was then initialized both from a default initialization and from the grid-search optimum. In all cases, the optimizer converged to the same maximum MLL value, indicating a stable and well-behaved marginal log-likelihood surface.

Table 3 reports the marginal log-likelihood values, together with the corresponding hyperparameter estimates obtained by coarse grid search and L-BFGS-B optimization for all models considered in Sections B.1 and B.2. Here, MLL denotes the marginal log-likelihood evaluated at the optimized hyperparameters.

	Grid search				L-BFGS-B			
Model	$\hat{\ell}$	$\widehat{\sigma_f^2}$	$\widehat{\sigma_y^2}$	MLL	$\hat{\ell}$	$\widehat{\sigma_f^2}$	$\widehat{\sigma_y^2}$	MLL
B.1(a)	0.6719	1.2022	0.1778	-15.8498	0.7215	1.2090	0.1590	-15.7686
B.1(b)	0.0758	1.2022	0.0025	-26.3689	0.0788	1.1000	0.0025	-26.2409
B.1(c)	0.3602	0.8595	0.0025	10.0101	0.3658	0.9298	0.0025	10.0227
B.2.2	0.3083	0.8595	0.0062	0.5491	0.3004	0.7382	0.0046	0.7236

Table 3: Maximum marginal log-likelihood (MLL) values evaluated at the optimized hyperparameters, together with the corresponding estimates, obtained by coarse grid search and L-BFGS-B optimization.

5.2 Prior Sanity Check for Kernel Derivatives

To verify the correctness of the kernel derivatives used in the noisy-input Gaussian process model (Section B.2), we perform a prior sanity check as suggested in the assignment. Specifically, we draw joint samples from the Gaussian process prior over function values and derivatives,

$$\begin{bmatrix} f \\ f' \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K & K_1^\top \\ K_1 & K_2 \end{bmatrix}\right),$$

where $K_{ij} = k(x_i, x_j)$, $(K_1)_{ij} = \partial k(x_i, x_j)/\partial x_i$, and $(K_2)_{ij} = \partial^2 k(x_i, x_j)/(\partial x_i \, \partial x_j)$. Sampling is performed on a grid of 100 equally spaced values X with samples $X_i \in [-1, 1]$.

From the sampled function values $f(X)$, we compute a numerical approximation to the derivative using central finite differences,

$$f'_i \approx \tilde{f}'_i = \frac{f(X_{i+1}) - f(X_{i-1})}{X_{i+1} - X_{i-1}}, \qquad i = 2, \dots, n-1.$$

Figure 17 compares the analytically sampled derivative $f'(X)$ to the numerical approximation $\tilde{f}'(X)$. The two curves coincide closely over the interior of the domain. This confirms that the analytical kernel derivatives and the resulting joint covariance structure for (f, f') are implemented correctly.

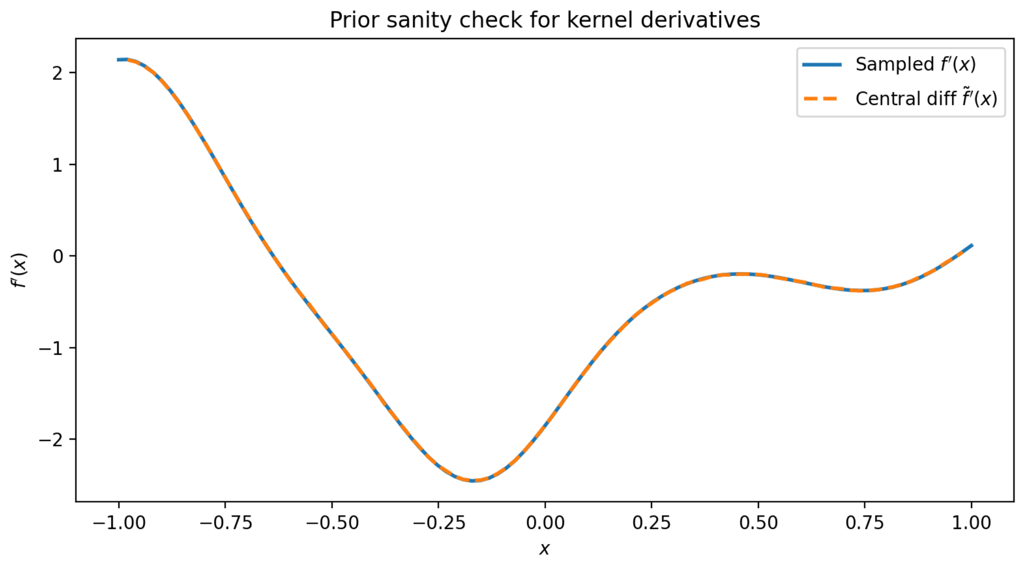


Figure 17: Prior sanity check for kernel derivatives. The derivative $f'(x)$ sampled directly from the joint GP prior over (f, f') (solid) is compared with a numerical central-difference approximation $\tilde{f}'(x)$ computed from the sampled function values $f(x)$ (dashed).

5.3 Bayesian Inference over Input Noise using NUTS

This appendix provides implementation details and diagnostics for the Bayesian inference over the latent input perturbations Δ described in Section B.2. We sample from the posterior

$$p(\Delta \mid X, y, \theta, \sigma_y^2, \sigma_x^2) \;\propto\; p(y \mid X, \Delta, \theta, \sigma_y^2)\, \mathcal{N}(\Delta; 0, \sigma_x^2 I),$$

where the likelihood covariance depends nonlinearly on Δ through C_Δ (Section B.2), rendering the posterior analytically intractable. Sampling is therefore performed using the No-U-Turn Sampler (NUTS).

Sampling was carried out using four parallel chains with 1500 warm-up iterations and 1500 retained samples per chain. Convergence was assessed using the $\hat{R}$ statistic and effective sample sizes (ESS), computed with ArviZ. All components satisfied $\hat{R} \approx 1$ and exhibited ESS values exceeding 200, indicating good mixing and convergence.

Figure 18 shows representative trace plots and marginal posterior densities for selected components of Δ. The chains mix well and show no pathological behavior. For the highlighted components, the posterior means are $\mathbb{E}[\Delta_{10}] \approx -0.254$ and $\mathbb{E}[\Delta_{11}] \approx 0.265$ with posterior standard deviations of approximately 0.05, closely matching the true values $(-0.25, 0.25)$.

The joint posterior samples of $(\Delta_{10}, \Delta_{11})$ in Figure 18 concentrate around the true values, indicating that the noisy-input GP model can recover the dominant input perturbations.

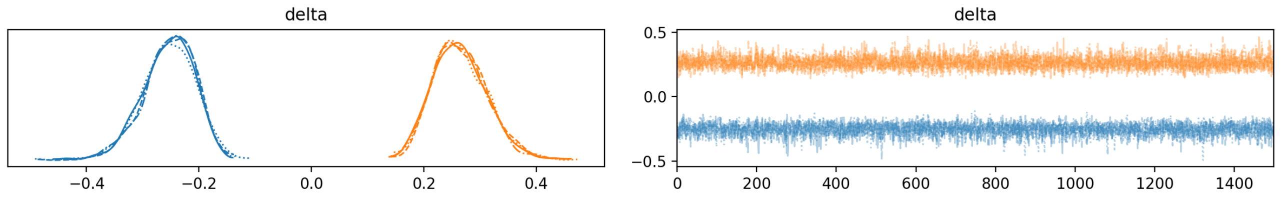


Figure 18: NUTS diagnostics for selected components of Δ. Left: marginal posterior densities. Right: trace plots across iterations. All chains show good mixing and convergence.

6 Generative AI Declaration

The signed Generative AI declaration is attached as a separate PDF on the next page.

UCPH's AI declaration

Declaration of using generative AI tools

☒ **I/we have used generative AI as an aid/tool** *(please tick)*

☐ **I/we have NOT used generative AI as an aid/tool** *(please tick)*

If generative AI is permitted in the exam, but you haven't used it in your exam paper, you just need to tick the box stating that you have not used GAI. You don't have to fill in the rest.

List which GAI tools you have used and include the link to the platform (if possible):

ChatGPT: https://chatgpt.com/
GitHub Copilot: https://github.com/copilot

Describe how generative AI has been used in the exam paper:

1) ***Purpose (what did you use the tool for?)***
   Generative AI was used to assist with coding (debugging), support idea generation, explanations of models and brainstorming.

2) ***Work phase (when in the process did you use GAI?)***
   The tools were used during different phases of the project, including initial idea development, programming, and later refinement of code before hand-in.

3) ***What did you do with the output? (including any editing of or continued work on the output)***
   Suggestions from generative AI were taken into account, and the final content reflects our own work and academic judgment.

Please note: Content generated by GAI that is used as a source in the paper requires correct use of quotation marks and source referencing. Read the guidelines from Copenhagen University Library at KUnet here.