PML FINAL PROJECT

Elizaveta Ignatova
spt315@alumni.ku.dk

Nicola Bertoni
wmf234@alumni.ku.dk

Giovanni Lombardi
nxm773@alumni.ku.dk

March 13, 2025

1 Introduction

All the code used for this project is available in our GitHub repository `https://github.com/GioLombardi/PML-Final-Project`. Our contributions were divided as follows:

Ignatova worked on the last two bullet points of Problem A.1 and contributed to the comparison of corresponding models (Sections 2.3, 2.4, and part of Section 3).

Bertoni contributed approximately 50% to the first two bullet points of Problem A.1 and comparison or related models (Sections 2.1, 2.2, and the first part of Section 3). Additionally, Bertoni contributed approximately 50% to Problem B (Section 4).

Lombardi contributed approximately 50% to Problem B and approximately 50% to the first two bullet points of Problem A.1 and comparison of the corresponding models.

2 Variations/extensions of the basic MNIST diffusion model (A.1)

2.1 Variation based on neural network predicting the mean

Idea. The first variation of the basic model we attempted to implement focused on having the neural network predict the mean rather than the noise score. To achieve this, we had to modify the function which reverses the process and so the loss function used during training. Following the approach described in the referenced article Ho et al. [2020], we formulated the loss for the single timestep $t-1$ as: $L_{t-1} = \mathbb{E}_q \left[\frac{1}{2\sigma_t^2} \left\| \tilde{\mu}(\mathbf{x}_t, \mathbf{x}_0) - \mu_\theta(\mathbf{x}_t, t) \right\|^2 \right] + C$. We then proceeded similarly to the basic model by sampling a single uniform timestep t (see Appendix B).

Conclusions. The implementation of this variation did not yield optimal results, and the generated samples did not meet our expectations. One potential reason for this outcome is that the ScoreNet class was specifically designed for the standard model and may not be well-suited for this variation.

2.2 Variations based on a different estimation of the ELBO

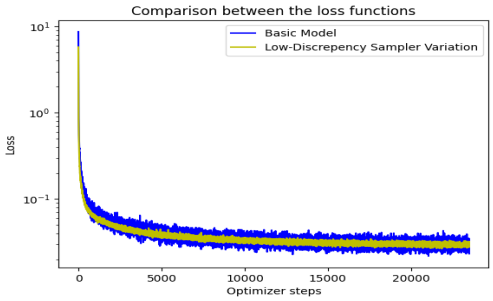
In the basic diffusion model we approximate several expectations with a single-sample Monte Carlo estimate, which results in rather noisy estimates for the ELBO. To address this issue, we explored two alternative approaches for the loss estimations, in order to reduce variance. Lower variance can facilitate faster optimization and lead to improved models.

Low-Discrepency Sampler

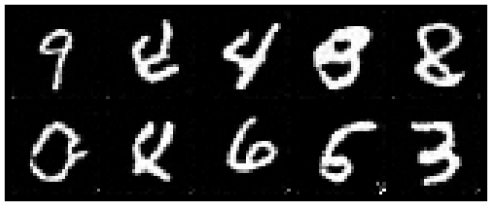
Idea. Our initial approach was inspired by the method described in the "Low-discrepancy sampler" section (Appendix I.1) of the *Variational Diffusion Models* paper Kingma et al. [2021]. The concept is straightforward: in the basic model, when processing a batch of data, timesteps are sampled independently from a uniform distribution. Instead, we modified this approach by sampling timesteps for the entire batch using a single uniform random number. While the original paper presents this method for the continuous case, we adapted it for the discrete case. As a result of this

implementation, each timestep now maintains a uniform marginal distribution, and the batch of timesteps more evenly covers the range $[0, 1]$ compared to independent sampling.
For this variation the only relevant change we made in the code was to insert a new function in the model class and then call it inside the elbo function already defined (Appendix A).

Results and Conclusions. After training the modified model, we observed that the generated samples (Figure 1b) were still of good quality, though there was no noticeable improvement compared to the basic model. However, we achieved a significant reduction in the variance of the loss estimate, as demonstrated in Figure 1a.



(a) comparison between the loss functions of the two models



(b) 10 samples generated by the LDs variation

Figure 1: Comparison and samples

Importance Sampling

Idea. In our second approach, we utilized importance sampling to select the timesteps, as proposed in the paper *Improved Denoising Diffusion Probabilistic Models* Nichol and Dhariwal [2021]. The core idea involves implementing an importance sampling algorithm for the timestep (t in the code), following the outline presented in the article: $L = \mathbb{E}_{t\sim\{p_t\}_t}\left[L_t/p_t\right]$, where $p_t = \sqrt{\mathbb{E}[L_t^2]}$ and $\sum p_t = 1$. To achieve this, we maintained a history of the last 10 values for each loss term (per timestep) and updated them dynamically during training. Initially, we sampled the timesteps uniformly, as done in the standard model, and continued this process until we had collected 10 samples for each timestep.
For this second variation, we first introduced two functions in the new model class with the purpose to update dinamically the loss history and compute the new normalized weights for importance sampling; then we used these two in a new elbo function we defined (see Appendix C).

Results and Conclusions. In this case as well, the generated samples (Figure 7b) were of good quality, but the results did not surpass those of the original model. Furthermore, although we initially believed that this new implementation would significantly reduce the variance of the loss estimate, it did not yield better results compared to the previous variation (Figure 7a). Specifically, the standard deviations of the loss values for each of the three models considered are:

	Basic model	**Model with LDs**	**Model with IS**
std	0.0959	0.0645	0.0727

2.3 Continuous-Time Diffusion Model Using SDE

The continuous-time SDE-based diffusion model introduces key improvements to enhance training, sampling, and overall performance.

Marginal Probability Standard Deviation. The function `marginal_prob_std` explicitly calculates the marginal probability's standard deviation using the parameter σ, providing better control over noise scheduling:

```
def marginal_prob_std(t, sigma=25.):
    return torch.sqrt((sigma**(2 * t) - 1.) / (2. * np.log(sigma)))
```

Diffusion Coefficient. `diffusion_coeff` models the time-dependent diffusion coefficient, controlling noise dynamics during forward diffusion:

```
def diffusion_coeff(t, sigma=25.):
    return sigma**t
```

Sampling with Euler-Maruyama. The SDE model uses the Euler-Maruyama method for efficient simulation of stochastic differential equations, ensuring faster, reliable sampling. See appendix for implementation

Training Process. Training minimizes the difference between the perturbed input and the model's output using the marginal probability in the loss function:

```
loss = torch.mean((score * marginal_prob_std(t)[:, None, None, None] + noise)**2)
```

As a result we got

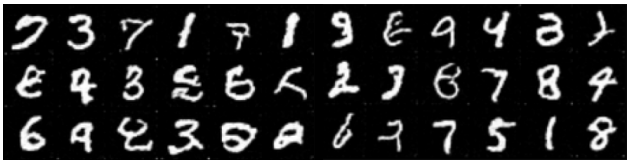


Figure 2: Generated samples with SDE.

Experimentation with Continuous Diffusion Models. The continuous formulation of diffusion models provides flexibility in defining noising processes and sampling methods.

Noising Processes. Two alternative approaches were explored:

1. **Logarithmic Noise Scale:** Introduces larger noise at earlier steps, enhancing gradient learning.
2. **Exponential Noise Decay:** Encourages smoother transitions with exponential noise reduction.

```
def marginal_prob_std_log(t, sigma=50.):
    return torch.sqrt((np.log(1 + sigma**t) - 1.) / np.log(sigma))
```

ODE Sampling. The reverse process is approximated with an ODE, providing deterministic, high-quality outputs at reduced computational cost (see attachment). ODE-based sampling generates sharper images, as shown in figure 9.

2.4 Conditional Sample Generation with Classifier-Free Guidance

The Classifier-Free Guidance approach enables class-conditional sample generation without requiring a pre-trained classifierDieleman [2022]. It alternates between conditional and unconditional training, and the guided score is computed as: $s_{\text{guided}} = (1 + w) \cdot s_\theta(x, t, y) - w \cdot s_\theta(x, t)$.
Key implementation:

```
guided_score = (1 + self.guidance_scale) * cond_score - self.guidance_scale *
↪  uncond_score
```

Results and Observations. Figure 3 demonstrates the results of classifier-free guided diffusion. The approach achieves high-quality class-specific generations, leveraging guidance without compromising diversity.

3 Quantitative comparisons

After implementing new variations/extensions of the basic model, we wanted to compare them to understand if some relevant improvements were achievied. In order to do this, we first made a comparison based on likelihood and on two metrics: Inception Score and FID score.

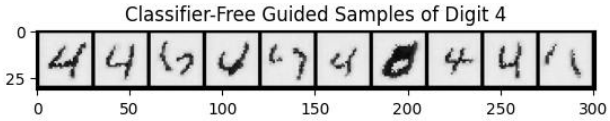


Figure 3: Classifier-Free Guided Diffusion.

Likelihood. As described in the article Ho et al. [2020], the $\text{ELBO}_{\text{simple}}$ is basically a very simple approximation of the log-likelihood $\mathbb{E}_{x_0 \sim q(x_0)}[-\log p_\theta(x_0)]$, which can be used in a very effective way to train the diffusion model. To evaluate the models, we aim to achieve the best possible approximation of the log-likelihood. Based on the previously mentioned article, we found that this approximation is given by the following formula:

$$\text{ELBO} = -L = -\mathbb{E}_{x_0 \sim q(x_0)}\left[D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)\|p(\mathbf{x}_T)) + \sum_{t>1} D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)) - \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right].$$

This formula only gives as an output a lower bound of the precise log-likelihood, which could also be meaningless to our purpose of comparison. We decided to compute these results both for the test and the train dataset, and draw our conclusion afterwards.

Inception score. The Inception Score (IS) is a metric used to assess the quality and diversity of images generated by generative models. It could be also highly useful for comparing different models. Following the definition we found in the article *A Note on the Inception Score* Barratt and Sharma [2018], given $x \sim p_{\text{gen}}$ a fixed number of samples generated by the trained model, the IS score is given by: $\text{IS} = \exp\left(\mathbb{E}_{x \sim p_{\text{gen}}}\left[D_{\text{KL}} p(y|x)\|p(y)\right]\right)$, where $p(y|\text{x})$ is the conditional class distribution. For this reason, we implemented our own Classifier model and trained it, obtaining a pretty high accuracy ($\geq 99\%$) (Appendix D) and then we defined a function which returns the IS score (Appendix E)

FID. The Fréchet Inception Distance (FID) is another metric used to evaluate generative models. Unlike the Inception Score, FID compares the distribution of generated images to the distribution of real images, rather than focusing solely on the distribution of the generated images. Given x_r, x_g two samples of real, respectively generated data, the idea is to extract their corresponding features and compute the following distance: $\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2\left(\Sigma_r \Sigma_g\right)^{1/2}\right)$.
Furthermore, we hypothesized that the two different estimations of the ELBO might help the model train more quickly. To verify this, we recorded the training time for each model.

Results and Conclusions

We chose to implement the functions for calculating each metric score ourselves and obtained the following results for the variations we developed.

Table 1: Comparison among first variations of the model

Model	**ELBO Test**	**ELBO Train**	**Inception Score**	**FID**	**Training time (s)**
DDPM	$1.7352 \cdot 10^3$	$1.7355 \cdot 10^3$	7.2212 ± 0.1919	8.1010	$2.7773 \cdot 10^3$
DDPM LDs	$1.7360 \cdot 10^3$	$1.7348 \cdot 10^3$	7.2426 ± 0.1835	8.8860	$2.7572 \cdot 10^3$
DDPM IS	$1.7411 \cdot 10^3$	$1.7409 \cdot 10^3$	7.3086 ± 0.2025	10.3433	$3.0524 \cdot 10^3$
SDE	$1.1250 \cdot 10^3$		7.15 ± 0.21	9.10	
SDE_2	$1.178 \cdot 10^3$		7.35 ± 0.22	8.95	
Conditional Sample Generation	$1.0500 \cdot 10^3$		6.70 ± 0.20	6.50	

Basing our final conclusions on the three possibile evaluation scores we introduced before, Table 1 shows that there aren't really meaningful changes between the basic model and the two models which differentiate the way to sample timesteps. Indeed, even if we managed to reduce the variance of the loss during the training, the quality of the samples generated are quite similar. For these specific models, we couldn't conclude anything in particular about the correlation between the log-likelihood and the quality of the model. Moreover, we observed the training speed of the model and concluded that it is not particularly remarkable, as it can also be influenced by various external factors.

In addiction, we put in Appendix H, some final visual comparisons among generated samples for each model.

4 Problem B: Function fitting with Constraints

4.1 Problem B.1: Fitting a standard GP

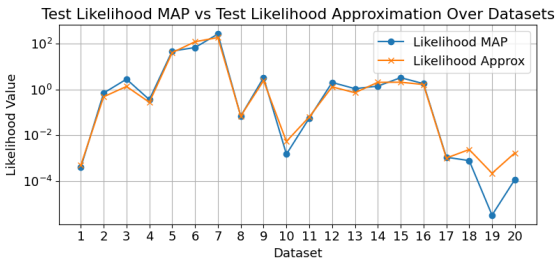
Choice of kernel and priors on parameters. For our Gaussian process, we decided to use a periodic kernel, $k(x, x') = \sigma^2 \exp\left(-\frac{2}{\ell^2}\sin^2\left(\pi\frac{|x-x'|}{p}\right)\right)$, which depends on three parameters: lengthscale ℓ, variance σ, and period p. This choice reflects the dataset's clear sinusoidal behavior. We also considered adding to this kernel a polynomial or RBF kernel, with the thought that adding this component could enhance the model's ability to learn the polynomial trend of the dataset. Ultimately, we opted for the simpler model for ease of training. For the parameter priors, we assigned log-normal distributions with mean 0 and variance 1 to both ℓ and σ, ensuring positivity. The selection of the mean and variance was arbitrary; we opted for these values because they seemed reasonable in the absence of any specific, obvious ideas. For the parameter p, we used a log-normal distribution with mean 0 and variance 1, shifted by 1 to ensure $p > 1$. That is, $p = 1 + \exp(Z)$, $Z \sim \mathcal{N}(0, 1)$. This decision was made because g is not periodic over the interval [0,1]. Therefore, it is reasonable to require the process trajectories to have a period greater than 1.

Computing the MAP estimate. Given the training set $\mathcal{D}_{\text{train}} = (X, y)$, we computed the MAP estimate θ^* of the model parameters by minimizing the negative joint log-likelihood with SGD. We computed the joint log-likelihood as $-\log p(y, \theta \mid X) = -\log p(y \mid \theta, X) - \log p(\theta)$. This approach is feasible because both the predictive and prior distributions are known (we chose the priors ourselves and we derived the predictive in class). Minimizing this expression is equivalent to minimizing the negative log-posterior: $-\log p(\theta|\mathcal{D}_{\text{train}}) = -\log p(y, \theta \mid X) + \log p(y \mid X)$ (using Bayes' theorem). We assessed the quality of the estimate by visually inspecting the predictive distribution $p(y_{\text{test}}|X_{\text{test}}, \mathcal{D}_{\text{train}}, \theta^*)$ across different dataset choices. Further details on the optimization procedure, along with a discussion of the encountered difficulties, are provided in Appendix I.

Sampling from the Posterior Distribution. We sampled from the posterior distribution using the No-U-Turn Sampler. We found that reparameterizing the model parameters to their unconstrained versions: $\tilde{\ell} = \log \ell, \tilde{\sigma} = \log \sigma, \tilde{p} = \log(p - 1)$, greatly enhanced sampling quality, resulting in better chain mixing and higher effective sample size values. With these reparameterized parameters, ArviZ diagnostics indicated good convergence and mixing, suggesting that NUTS provided reliable samples from the posterior. The ArviZ summary and plots are shown in Appendix J, along with a comparison to sampling using the constrained (original) parameters. After obtaining $N = 500$ samples $\theta_1, \ldots, \theta_N$ from the posterior, we approximated the predictive distribution for the test data using Monte Carlo integration:

$$p(y_{\text{test}}|X_{\text{test}}, \mathcal{D}_{\text{train}}) = \int p(y_{\text{test}}|X_{\text{test}}, \mathcal{D}_{\text{train}}, \theta)p(\theta|\mathcal{D}_{\text{train}})\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} p(y_{\text{test}}|X_{\text{test}}, \mathcal{D}_{\text{train}}, \theta_i). \tag{1}$$

Comparison of test likelihoods. We evaluated the test likelihoods on 20 random datasets using two methods: computing $p(y_{\text{test}}|X_{\text{test}}, \mathcal{D}_{\text{train}}, \theta^*)$ with the MAP estimate θ^* and approximating it using 500 posterior samples from NUTS as in equation (1). Figure 10 presents the results for each dataset. Both methods yielded consistently similar outcomes, with notable differences only in datasets 19 and 20. The slight improvement using NUTS in these challenging cases (in which both methods performed poorly) is likely due to accounting for hyperparameter uncertainty. By integrating over the posterior distribution of θ, we capture a range of plausible parameters, enhancing predictive performance when the MAP estimate is unreliable.



(a) Plot of test likelihoods over datasets

Method	mean	std
MAP	19.49	57.61
Approximate	17.70	46.22

(b) Mean and std for both methods

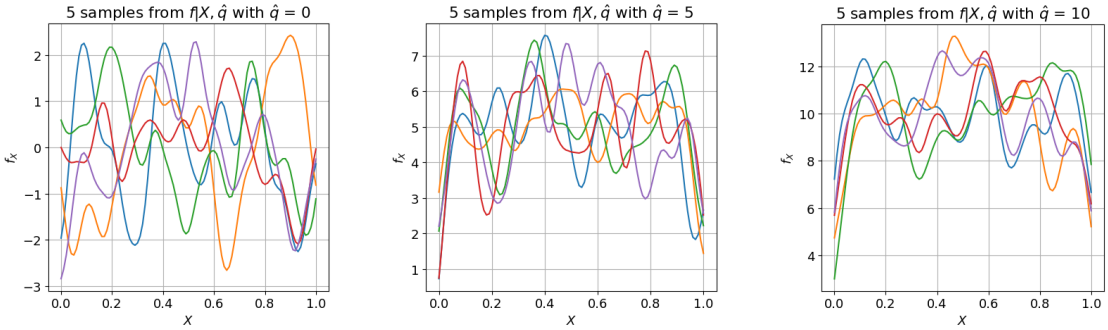
Figure 4: Comparison of test likelihoods

4.2 Problem B.2: Learning with Integral Constraints

Distribution of the constrained finite-dimensional marginals. From now on, we fix the parameters of the model to a MAP estimate obtained in B.1. To derive the distribution of $(\hat{q}, f)|X$, define $f = [f(x_1), \ldots, f(x_\ell)]^T$ and $w = [w_1, \ldots, w_\ell]^T$. Write $\hat{q} = \sum_{i=1}^{\ell} w_i f(x_i) = w^T f$. It follows that $\begin{bmatrix} \hat{q} \\ f \end{bmatrix} = \begin{bmatrix} w^T \\ I_\ell \end{bmatrix} f$, where I_ℓ denotes the identity matrix of order ℓ. Since $f \sim \mathcal{N}(0, K)$, where $K_{ij} = k(x_i, x_j)$, and k is the kernel of the model, by the properties of the normal distribution (see e.g., Oswin's notes, pag. 11), we get

$$\begin{bmatrix} \hat{q} \\ f \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} w^T \\ I_\ell \end{bmatrix} K \begin{bmatrix} w & I_\ell \end{bmatrix}\right) = \mathcal{N}\left(0, \begin{bmatrix} w^T K w & w^T K \\ K w & K \end{bmatrix}\right).$$

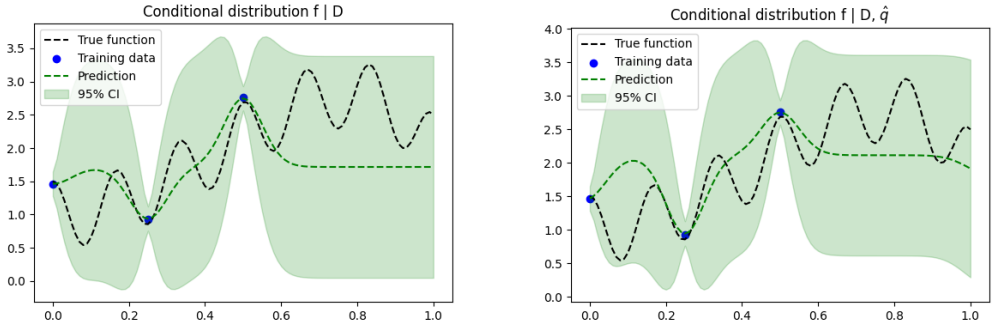
Since the vector $(\hat{q}, f)|X$ is normally distributed, we can compute $f|X, \hat{q}$ using the properties of the normal distribution. Specifically, using equation (29) on pag. 9 of Oswin's notes we get $f|X, \hat{q} \sim \mathcal{N}(\mu, \Sigma)$ with $\mu = Kw(w^T K w)^{-1}\hat{q}$ and $\Sigma = K - Kw(w^T K w)^{-1} w^T K$. The covariance matrix is not full rank. Indeed, the vector w belongs to its null space: $\Sigma w = Kw - Kw(w^T K w)^{-1}(w^T K w) = Kw - Kw = 0$. To sample from this distribution, we added a small jitter term of 10^{-5} to ensure positive definiteness. Figure 5 presents samples from $f \mid X, \hat{q}$ for various values of $\hat{q}$. As expected, the trajectories align with the integral constraint. Notably, the samples consistently tend toward zero near the domain edges. A mathematical explanation of this phenomenon is provided in Appendix K.



(a) $\hat{q} = 0$ (b) $\hat{q} = 5$ (c) $\hat{q} = 10$

Figure 5: Samples from the GP for different integral values $\hat{q}$

Posterior distributions with and without integral constraint. Given the dataset $\mathcal{D}$ and the integral value $\hat{q}$ provided in the assignment, we visually compare in Figure 6 the posterior distributions of $f \mid \mathcal{D}$ and $f \mid \mathcal{D}, \hat{q}$. Computing the latter required a mathematical derivation, that is illustrated in Appendix L. The plots are not very informative: the mean values do not align closely with the true function g, and the marginal variances are large in both cases. We believe this is because $\mathcal{D}$ contains only three data points. In Appendix M, we compare the same distributions using richer datasets and observe improved predictions due to the integral constraint.



(a) $\hat{q} = 0$ (b) $\hat{q} = 5$

Figure 6: Comparison of test likelihoods

References

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf.

Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 21696–21707. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/b578f2a52a0229873fefc2a4b06377fa-Paper.pdf.

Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8162–8171. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/nichol21a.html.

Sander Dieleman. Guidance: a cheat code for diffusion models, 2022. URL https://benanne.github.io/2022/05/26/guidance.html.

Shane Barratt and Rishi Sharma. A note on the inception score, 2018. URL https://arxiv.org/abs/1801.01973.

Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, January 2006. URL https://www.microsoft.com/en-us/research/publication/pattern-recognition-machine-learning/.

A Implementation of Low-Discrepency Sampler

The following one is the only new function we defined for this new version of the model.

```
def low_discrepancy_sampler(self, batch_size):
    u_0 = torch.rand(1).item()
    t = torch.fmod(u_0 + torch.arange(0, batch_size, dtype=torch.float32) / batch_size,
    ↪  1.0)
    t = (t * self.T).long() + 1
    return t.unsqueeze(-1).to(self.beta.device)
```

The function first samples a random number $u_0 \sim U[0, 1]$. It then creates a tensor with the same number of elements as the batch size, where the values are uniformly distributed between u_0 and $u_0 + 1$. Next, it wraps these values into the range $[0, 1)$ by computing the remainder when each value is divided by 1. Finally, the function multiplies each element of the tensor by T and rounds the results to the nearest integer (because of the discrete model we are taking).

B Implementation of the model where the neural network predict the mean

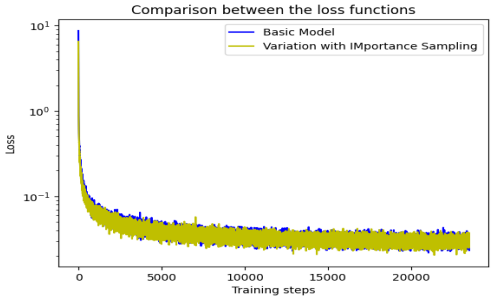
Even if this code we implemented didn't generate samples like we wanted, we put the 2 main variations we made from the original one in this Appendix.

```
    def reverse_diffusion(self, xt, t, epsilon):
        mean =  self.network(xt, t)

        std1 = torch.where(t>0, torch.sqrt(((1-self.alpha_bar[t-1]) /
        ↪  (1-self.alpha_bar[t])) * self.beta[t]), 0)
        std2 =  torch.where(t>0, torch.sqrt(self.beta[t]), 0)

        return mean + std2 * epsilon

def elbo_mean(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)

```

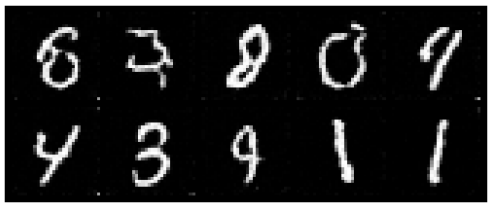
```
    #return (-nn.MSELoss(reduction='mean')(mean_t1, self.network(xt, t))) #* (1 /
    ↪  (2*(std1**2)))

    # We compute 2 different means at the time t (respectively using Equation (7) and
    ↪  Equation (10) from the paper)

    coef1 = (torch.sqrt(self.alpha_bar[t-1]) * self.beta[t]) / (1-self.alpha_bar[t])
    coef2 = (torch.sqrt(self.alpha[t]) * (1-self.alpha_bar[t-1])) / (1-self.alpha_bar[t])
    mean_t1 = coef1 * x0 + coef2 * xt

    mean_t2 = (1/torch.sqrt(self.alpha[t])) * (xt - ((self.beta[t] * epsilon) /
    ↪  torch.sqrt(1 - self.alpha_bar[t])))

    std1 = torch.where(t>0, torch.sqrt(((1-self.alpha_bar[t-1]) / (1-self.alpha_bar[t]))
    ↪  * self.beta[t]), 0)
    std2 =  torch.where(t>0, torch.sqrt(self.beta[t]), 0)

    l = (-nn.MSELoss(reduction='none')(mean_t1, self.network(xt, t))) * (1 /
    ↪  (2*(std1**2)))
    return l.mean()


```

C Figures and Implementation for the IS model

In the following figures, we illustrate how the variance of the loss improved for the new model, along with 10 samples it generated.



(a) Comparison between the loss functions of the two models



(b) 10 samples generated by the IS variation

Figure 7: comparison and samples

Here we also put the functions we defined in the new Model Class for the Importance Sampling variation.

```
self.loss_history = torch.zeros((T, 10), device=self.beta.device)

def update_loss_history(self, t, loss):
    for i, step in enumerate(t.squeeze().tolist()):
        self.loss_history[step - 1] = torch.cat((self.loss_history[step - 1, 1:],
        ↪  loss[i].unsqueeze(0).to(self.loss_history.device))).to(self.loss_history.device)

def compute_importance_weights(self):
    mean_squared_losses = (self.loss_history**2).mean(dim=1)
    weights = torch.sqrt(mean_squared_losses)
    normalized_weights =weights / weights.sum()
    return normalized_weights

```

```
def elbo_IS(self, x0, step=0):
    if  torch.any(self.loss_history == 0).item():
        t = torch.randint(1, self.T, (x0.shape[0], 1)).to(x0.device)
        weights = None
    else:
        weights = self.compute_importance_weights()
        t = torch.multinomial(weights, x0.shape[0], replacement=True).unsqueeze(-1) + 1

    p_t = weights[t.squeeze() - 1] if weights is not None else 1 / self.T

    epsilon = torch.randn_like(x0)

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

    per_sample_loss = nn.MSELoss(reduction='none')(epsilon, self.network(xt,
    ↪  t)).mean(dim=1)
    reweighted_loss = (per_sample_loss / p_t).mean() if weights is not None else
    ↪  per_sample_loss.mean()

    self.update_loss_history(t, per_sample_loss.detach())

    return -reweighted_loss
```

D Implementation of a MNIST classifier

As we already mentioned, in order to compute the Inception Score and the FID score, we implemented our own MNIST classifier, we trained it, and we observed the accuracy obtained. The following is the model we implemented.

```
from torch.optim.lr_scheduler import StepLR

cnn_model = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=7, padding=3),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),  # The shape becomes (14, 14)
    nn.Dropout(0.4),
    nn.Conv2d(32, 64, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),  # The shape becomes (7, 7)
    nn.Dropout(0.4),
    nn.Conv2d(64, 64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),  # The shape becomes (4, 4)
    nn.ReLU(),
    nn.AvgPool2d(4),
    nn.Flatten(),
    nn.Linear(128, 10),
)
cnn_model.to(device)
print(cnn_model)

```

Here we provide also an Example of usage of the classifier, made on samples generated by the DDPM model:

E Implementation of IS and FID functions

In this Appendix, we inserted the codes for the two functions we used for evalutaing the IS and FID metrics of every model: both of them require a classifier to be defined.

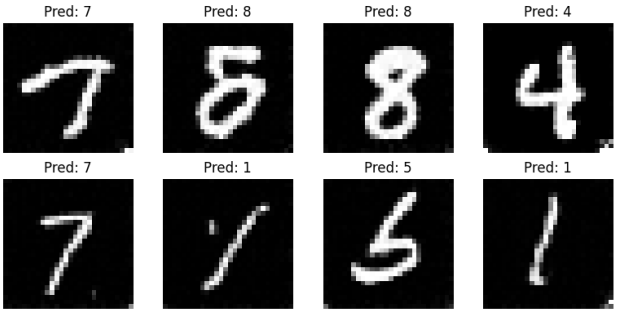


Figure 8

```python
def inception_score(samples, classifier, splits=50):
    classifier.eval()
    with torch.no_grad():
        probs = F.softmax(classifier(samples), dim=1).cpu().numpy()
    # Split the probabilities into chunks
    split_scores = []
    N = len(probs)
    for k in range(splits):
        part = probs[k * (N // splits): (k + 1) * (N // splits), :]
        p_y = np.mean(part, axis=0)
        kl_divs = part * (np.log(part + 1e-6) - np.log(p_y + 1e-6))
        split_scores.append(np.exp(np.mean(np.sum(kl_divs, axis=1))))

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

def calculate_fid(real_features, generated_features):
    mu_real = np.mean(real_features, axis=0)
    mu_gen = np.mean(generated_features, axis=0)
    sigma_real = np.cov(real_features, rowvar=False)
    sigma_gen = np.cov(generated_features, rowvar=False)

    diff = mu_real - mu_gen
    covmean, _ = sqrtm(sigma_real @ sigma_gen, disp=False)

    fid = np.linalg.norm(diff) + np.trace(sigma_real + sigma_gen - 2 * covmean)
    return fid
```

F Sampling with Euler-Maruyama

```python
def Euler_Maruyama_sampler(score_model, marginal_prob_std, diffusion_coeff,
    batch_size=256, num_steps=1000, device='cuda', eps=1e-3):
    t = torch.ones(batch_size, device=device)
    init_x = torch.randn(batch_size, 1, 28, 28, device=device) * marginal_prob_std(t)[:,
        None, None, None]
    time_steps = torch.linspace(1., eps, num_steps, device=device)
    step_size = time_steps[0] - time_steps[1]
    x = init_x
    with torch.no_grad():
        for time_step in tqdm(time_steps):
            batch_time_step = torch.ones(batch_size, device=device) * time_step
            g = diffusion_coeff(batch_time_step)
            mean_x = x + (g**2)[:, None, None, None] * score_model(x, batch_time_step) *
                step_size
```

```
            x = mean_x + torch.sqrt(step_size) * g[:, None, None, None] *
            ↪  torch.randn_like(x)
    return x

```

G Implementation of ODE-based deterministic reverse process

```
def ode_sampler_experiment(score_model, marginal_prob_std, batch_size=256,
                           device='cuda', num_steps=500):
    t = torch.ones(batch_size, device=device)
    init_x = torch.randn(batch_size, 1, 28, 28, device=device) \
              * marginal_prob_std(t)[:, None, None, None]

    def ode_func(t, x_flat):
        x = torch.tensor(x_flat, device=device).view(batch_size, 1, 28, 28)
        time_tensor = torch.tensor([t], device=device)
        g = diffusion_coeff_exp(time_tensor)
        dx_dt = -g**2 * score_model(x, time_tensor).view(-1)
        return dx_dt.cpu().numpy()

    time_points = np.linspace(1.0, 1e-3, num_steps)
    x_flat = init_x.view(-1).cpu().numpy()
    solution = solve_ivp(ode_func, (1.0, 1e-3), x_flat, t_eval=time_points,
    ↪  method='RK45')
    x_samples = torch.tensor(solution.y[:, -1], device=device).view(batch_size, 1, 28,
    ↪  28)
    return x_samples
```

Figure 9: SDE2

H Visual Comparisons

Here we put 36 samples for every model we implemented, including the standard model, in order to observe the quality and diversity of each result. We did not include samples for the model described in 2.1 because they were not sufficiently representative; we only achieved small improvements by significantly increasing the number of epochs, but it was still not enough.

(a) Samples generated by the basic model

(b) Samples generated by the LDs variation

(c) Samples generated by the IS variation

(d) Samples generated by the SDE variation

Figure 10

I Optimization of the negative joint log-likelihood

To optimize the negative joint log-likelihood $-\log p(y, \theta \mid X)$, we used the Adam optimizer (which implements a variation of SGD) with a fixed learning rate of 10^{-2} and $3 \cdot 10^3$ optimization steps. To prevent the optimizer from updating parameters outside their domain, we reparametrized ℓ, σ, and p using their unconstrained versions:

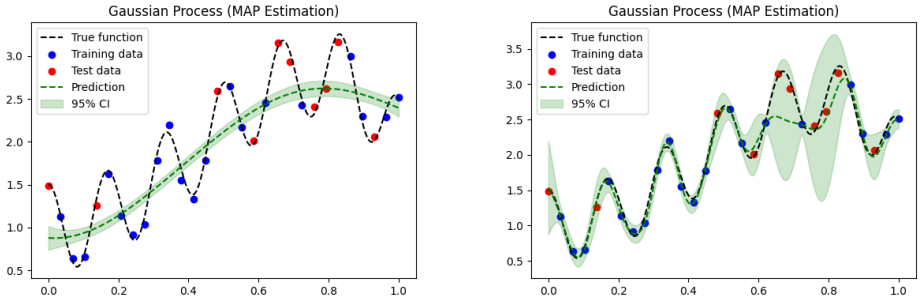
$$\tilde{\ell} = \log \ell, \quad \tilde{\sigma} = \log \sigma, \quad \tilde{p} = \log(p - 1).$$

The Adam optimizer then optimized these variables.

With these settings, the optimization generally yielded satisfactory results. However, for some rare dataset choices, the optimization would fail, resulting in the GP learning only the general trend of the data but not the sinusoidal behavior. To address this, we restarted the optimizer five times with different initial parameter values and retained the best result. The parameters were initially set to 0, then to 0.5, -0.5, 1, and finally -1. This approach consistently produced good results across all datasets we encountered, enabling the GP to learn the sinusoidal behavior.

An example is shown in Figure 11. The figure illustrates the predictive distribution $p(y_{\text{pred}}|X_{\text{pred}}, \mathcal{D}_{\text{train}}, \theta^*)$ for the MAP estimate θ^* obtained by optimizing with initial parameter values set to 0 (on the left) and for the MAP estimate obtained after training with four restarts (on the right). The problematic dataset can be generated in the code by calling

```
x_train, y_train_raw, x_test, y_test_raw = generate_dataset(seed=12)
```



(a) Without restarts

(b) With restarts

Figure 11: An example of bad optimiazation

J Improving sampling efficiency with parameter reparameterization in NUTS

Table 2 presents selected statistics from the ArviZ summary of the sampling process for both the original parameters and their reparameterized, unconstrained versions. We observe a significant improvement when using the unconstrained parameters: the effective sample size values for σ and p increase substantially, and the potential scale reduction factor $\hat{R}$ decreases to its optimal value of 1.0. This indicates that the sampling with unconstrained parameters has converged well, and NUTS has provided meaningful and reliable samples.

Table 2: Some selected statistics from Arviz summaries

	Original			Reparametrized		
Parameter	ess_bulk	ess_tail	r_hat	ess_bulk	ess_tail	r_hat
ℓ	413	427	1.01	627	534	1.0
σ	266	243	1.02	482	445	1.0
p	267	186	1.02	465	555	1.0

This improvement is also evident when comparing trace plots (as we see in Figure 12): the Markov chains for the unconstrained parameters exhibit better mixing and more thorough exploration of the parameter space. By reparameterizing the model to transform constrained parameters into unconstrained variables, we allow the sampling algorithm to navigate the parameter space more effectively, enhancing sampling efficiency and convergence.

K Explanation of the unusual behavior of samples from the GP with integral constraint

In Figure 5, we observed that the samples from the Gaussian Process with integral constraints, given by the values $\hat{q} = 5$ and $\hat{q} = 10$, consistently tend toward zero near the domain edges. This appendix aims to explain this behavior.

Recall that the samples are drawn from $f|X, \hat{q} \sim \mathcal{N}(\mu, \Sigma)$, where

$$\mu = Kw(w^T Kw)^{-1}\hat{q},$$
$$\Sigma = K - Kw(w^T Kw)^{-1}w^T K.$$

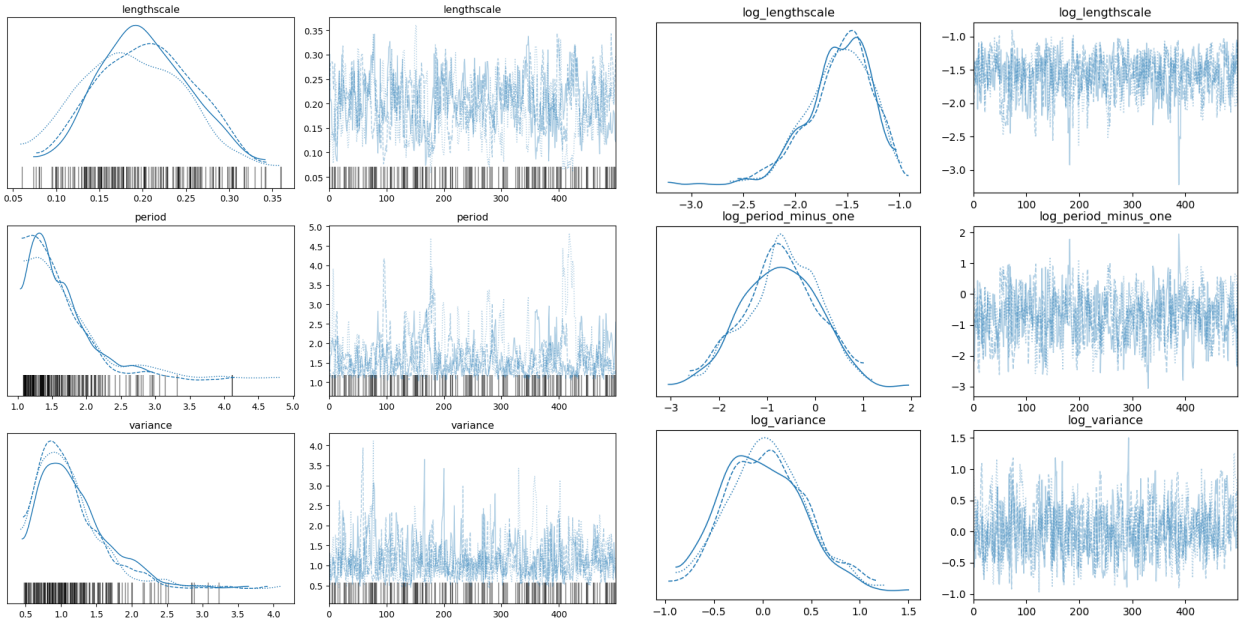
Note that μ contains only positive elements since both K and w have positive elements. We argue that the first and last elements of μ are the closest to zero.

Note that μ is proportional to the vector Kw,

$$\mu = Kw(w^T Kw)^{-1}\hat{q} \propto Kw.$$

Let $K_{i,:}$ denote the i-th row of K. Given that $\mu \propto Kw$, the i-th element μ_i of μ is proportional to $K_{i,:} \cdot w$ (with a positive constant that does not depend on i). This is essentially the trapezoidal rule approximation of the integral of the function

$$\phi_i : x \to k(x_i, x).$$



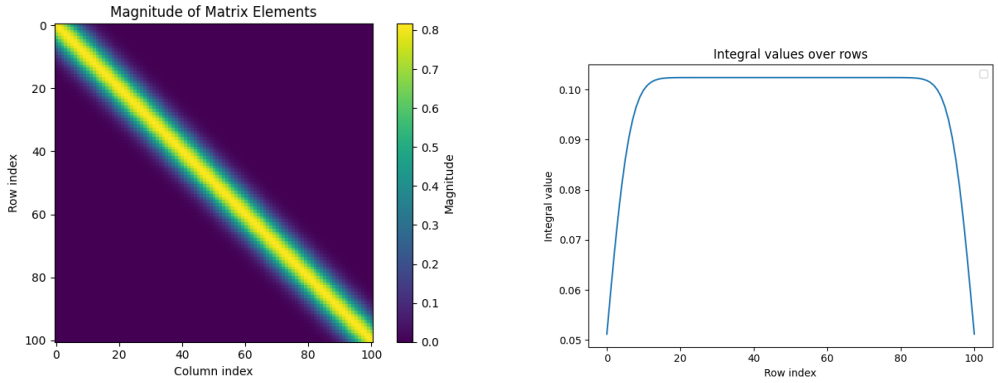
(a) For original/contrained parameters

(b) For reparametrized/uncontrained parameters

Figure 12: Arviz desity and trace plots

When the period parameter p satisfies $p > 2$, the value $k(x, y)$ decreases as the separation between x and y increases for every $x, y \in [0, 1]$. This is because $p > 2$ ensures that the argument of the sine function inside the kernel is always at most $\pi/2$. Crearly, the points at the extremes of the interval $[0, 1]$ are the furthest from other points within the interval. Therefore, we expect the integral of ϕ_i to be lowest when $i = 1$ (the smallest index) or $i = \ell$ (the largest index).

In our case, we only constrained $p > 1$, so generally p won't be greater than 2. However, for $p \approx 1.3$ (as in our practical scenario), the ϕ_i still tend to have a lower integral for i at the extremes. The underlying idea remains the same, although it is less immediate. Figure 13 illustrates the magnitude of the kernel values computed on a grid of $\ell = 101$ points in [0,1] using the parameters from the MAP estimate (on the left), and the approximate integral of the rows of K using the trapezoidal rule (on the right).



(a) Kernel matrix

(b) Intral values over rows

Figure 13: Kernel matrix illustration

L Derivation of the conditional distribution with integral constraint

Let's derive the distribution of $f|\hat{q}, \mathcal{D}$, using the properties of the Gaussian distribution. Denote $x = [x_1, \ldots, x_\ell]^T$ the x values for the integral approximation, $\tilde{x} = [\tilde{x}_1, \tilde{x}_2, \tilde{x}_3]^T$ the x values of the points in the given dataset $\mathcal{D}$

(although we illustrate with three data points, the same approach extends to datasets with more points). Denote also $f = [f(x_1), \ldots, f(x_\ell)]^T$, $\tilde{f} = [f(\tilde{x}_1), f(\tilde{x}_2), f(\tilde{x}_3)]^T$ and $w = [w_1, \ldots, w_\ell]^T$.

By the properties of the Gaussian process, we have:

$$\begin{bmatrix} \tilde{f} \\ f \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K_{\tilde{f}\tilde{f}} & K_{\tilde{f}f} \\ K_{f\tilde{f}} & K_{ff} \end{bmatrix}\right),$$

where

$$\begin{aligned}
(K_{\tilde{f}\tilde{f}})_{ij} &= k(\tilde{x}_i, \tilde{x}_j), \text{ for } i,j = 1,2,3, \\
(K_{\tilde{f}f})_{ij} &= k(\tilde{x}_i, x_j), \text{ for } i = 1,2,3,\ j = 1,\ldots,\ell, \\
(K_{f\tilde{f}})_{ij} &= k(x_i, \tilde{x}_j), \text{ for } j = 1,2,3,\ i = 1,\ldots,\ell, \\
(K_{ff})_{ij} &= k(x_i, x_j) \text{ for } i,j = 1,2,\ell.
\end{aligned}$$

Since the observed values y are distributes as $y = \tilde{f} + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma_y I_3)$ independent Gaussian noise, by a known property of the Gaussian distribution we get

$$\begin{bmatrix} y \\ f \end{bmatrix} = \begin{bmatrix} \tilde{f} \\ f \end{bmatrix} + \begin{bmatrix} \epsilon \\ 0 \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K_{yy} & K_{yf} \\ K_{fy} & K_{ff} \end{bmatrix}\right),$$

with

$$\begin{aligned}
K_{yy} &= K_{\tilde{f}\tilde{f}} + \sigma_y I_3, \\
K_{yf} &= K_{fy}^T = K_{\tilde{f}f} = K_{f\tilde{f}}^T.
\end{aligned}$$

Similarly as we saw in the main text, a straightforward computation shows that

$$\begin{bmatrix} y \\ \hat{q} \\ f \end{bmatrix} = \begin{bmatrix} I_3 & 0 \\ 0 & w^T \\ 0 & I_\ell \end{bmatrix} \begin{bmatrix} y \\ f \end{bmatrix},$$

and thus (using the closure under linear transformations of normally distributed random variables, see, e.g., Oswin's notes on pag.11)

$$\begin{bmatrix} y \\ \hat{q} \\ f \end{bmatrix} = \begin{bmatrix} I_3 & 0 \\ 0 & w^T \\ 0 & I_\ell \end{bmatrix} \begin{bmatrix} y \\ f \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K_{yy} & K_{yf}w & K_{yf} \\ w^T K_{fy} & w^T K_{ff} w & w^T K_{ff} \\ K_{fy} & K_{ff}w & K_{ff} \end{bmatrix}\right).$$

Introducing the notations

$$\begin{aligned}
K_{yq,yq} &= \begin{bmatrix} K_{yy} & K_{yf}w \\ w^T K_{fy} & w^T K_{ff} w \end{bmatrix}, \\
K_{yq,f} &= \begin{bmatrix} K_{yf} \\ w^T K_{ff} \end{bmatrix} = K_{f,yq}^T, \\
v_{yq} &= \begin{bmatrix} y \\ \hat{q} \end{bmatrix},
\end{aligned}$$

we have

$$\begin{bmatrix} y \\ \hat{q} \\ f \end{bmatrix} = \begin{bmatrix} v_{yq} \\ f \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K_{yq,yq} & K_{yq,f} \\ K_{f,yq} & K_{ff} \end{bmatrix}\right).$$

Thus, using the properties of the normal distribution (see, e.g., equation (29) on pag. 9 of Oswin's notes or Section 2.3.1 in Bishop [2006]), we have $f|\hat{q}, \mathcal{D} \sim \mathcal{N}(\mu, \Sigma)$ with

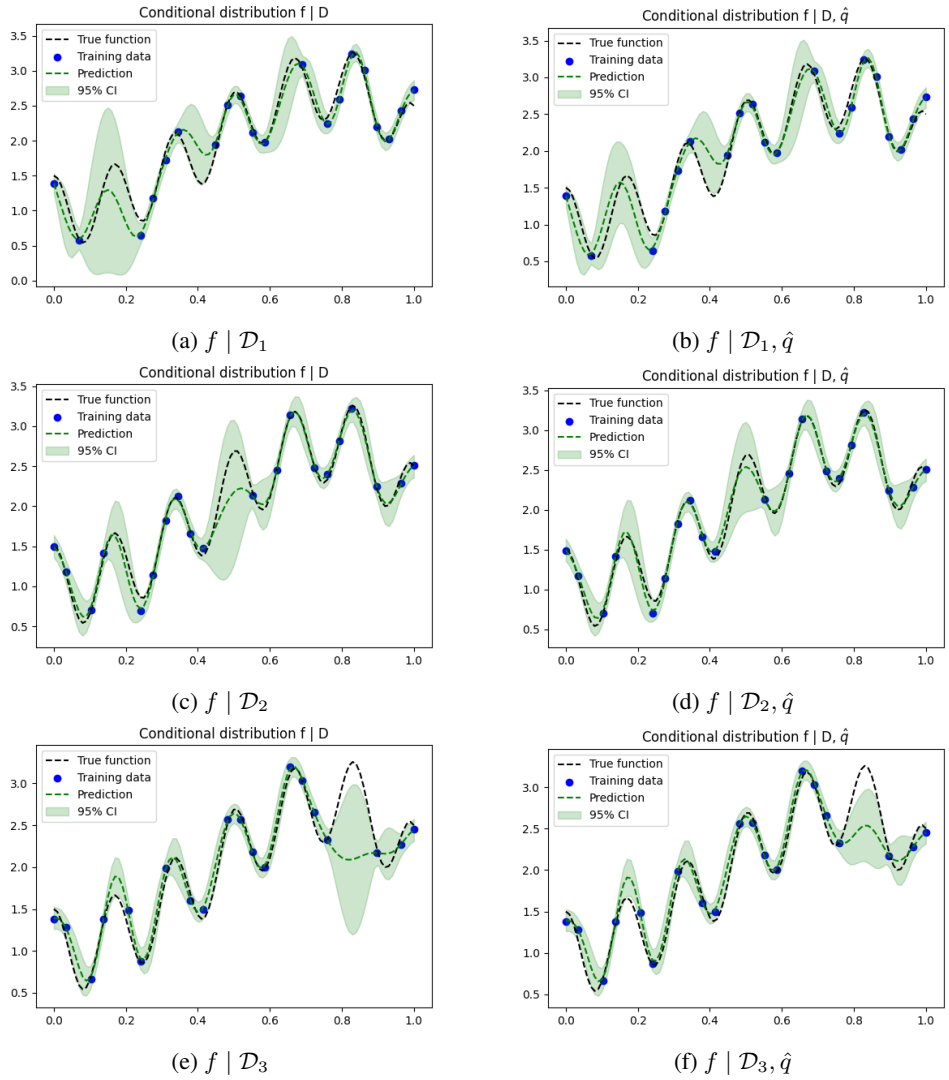
$$\begin{aligned}
\mu &= K_{f,yq} K_{yq,yq}^{-1} v_{yq}, \\
\Sigma &= K_{ff} - K_{f,yq} K_{yq,yq}^{-1} K_{yq,f}.
\end{aligned}$$

These computations were implemented in the method `conditional_integral_with_data` of the class `GaussianProcessIntegral`.

M Enhancing model predictions with integral constraints

To better appreciate the positive impact of introducing the integral constraint on the model's predictions, we selected three datasets $\mathcal{D}_i$, $i = 1, 2, 3$, constructed as in the first part of Problem B (but we only retained the training data points and descarded the test data) and we visually compared the distributions $f \mid \mathcal{D}_i$ and $f \mid \mathcal{D}_i, \hat{q}$, as requested in the last part of Problem B for the dataset $\mathcal{D}$. The model's parameters were fixed to reasonable values (a MAP estimate found in Problem B.1 for a random dataset).

The results are illustrated in Figure 14. We observe that adding the integral constraint improves the predictions: the means of the marginals align more closely with the ground truth function, and the variances decrease, particularly in regions where predictions were poor due to missing training data.



(a) $f \mid \mathcal{D}_1$ (b) $f \mid \mathcal{D}_1, \hat{q}$

(c) $f \mid \mathcal{D}_2$ (d) $f \mid \mathcal{D}_2, \hat{q}$

(e) $f \mid \mathcal{D}_3$ (f) $f \mid \mathcal{D}_3, \hat{q}$

Figure 14: Comparison of conditional distributions with and without integral constraint