# Take-home Exam

*Deadline:* January 11, 2026, 23:59

This document provides the instructions for the take-home exam of the course *Advanced Probabilistic Machine Learning* (SSY316).
The exam consists of three parts. For each part, the corresponding datasets are available on the course website. All parts of the exam must be completed and submitted.
Each part requires the submission of a short project-style report. The exam must be completed in groups of **two students**, according to the previously registered group list.
The total score for the exam is **40 points**, distributed as follows:

- **P1:** 10 points
- **P2:** 5 points
- **P3:** 25 points

## *Instructions for Writing Your Report and Submission*

- Submit your solutions to all the questions in this document as a single PDF file. Your report should include:
  - Figures and plots for visualization.
  - Important derivations and results.
  - Design choices and explanations of your implementation.

  Bibliography, simulation code, and additional material can be added as an appendix.
- For each question, submit a single Python script or Jupyter notebook file named `Pi.py` or `Pi.ipynb`, where $i = 1, 2, 3$.
- Create a `requirements.txt` file listing all necessary Python packages required.
- Place all your submission files (scripts, notebooks, and `requirements.txt`) in a zip file. Do not include the dataset in the zip file; assume the dataset is placed in the same directory as your script when executed.
- Ensure that your script or notebook can be executed without errors in a clean Python virtual environment ($\geq 3.8$). It should work with the following commands:

```
pip install -r requirements.txt
python Pi.py  # For Python scripts
jupyter notebook Pi.ipynb  # For Jupyter notebooks
```

- Your solution should be well-documented, with clear explanations and comments in the code.

## P1. Skill Estimation in Competitive Games with TrueSkill (10 points)

In this task, you will design and analyze a Bayesian model for estimating the skills of players based on match outcomes, using the TrueSkill ranking system developed by Microsoft Research. TrueSkill assigns a Gaussian-distributed skill to each player and updates these skills using Bayesian inference based on observed match results.
You will analyze match outcomes from the Italian 2018/2019 Serie A football league to explore how skill distributions evolve with data. The dataset, `SerieA.csv`, available on Canvas (SerieA.csv), contains the match results from this league. Additionally, you will extend the analysis by testing the method on a dataset of your choice.
The probabilistic model you will define represents all quantities as random variables. In this model:

- Each player's skill is a Gaussian random variable.
- When two players compete in a match, the outcome is modeled as a Gaussian random variable with a mean equal to the difference in the two players' skills.
- The result of the match is determined as follows:
- A result of 1 indicates the victory of Player 1 if the outcome is greater than zero.
- A result of -1 indicates the victory of Player 2 if the outcome is less than zero.

Your task is to use this model to estimate skills and analyze the performance of the ranking system based on the provided and selected datasets.

### P1.1. Model Formulation (1 points)

1. Define the Bayesian model for a single match:
   - **Skills:** Represent the skills of two players as Gaussian random variables $s_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $s_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$.
   - **Outcome:** Model the outcome of the match as a Gaussian random variable $t \sim \mathcal{N}(s_1 - s_2, \sigma_t^2)$.
   - **Result:** Define a discrete variable $y$ where $y = 1$ if $t > 0$ (Player 1 wins) and $y = -1$ if $t \leq 0$ (Player 2 wins).
2. Specify the joint distribution of all variables and identify the five hyperparameters in the model.

## P1.2. Bayesian Network and Factor Graph (1 points)

1. Draw the Bayesian network for the model.
2. Convert the network into a factor graph with:
   - Variable nodes for $s_1, s_2, t, y$.
   - Factor nodes for priors, skill differences, and the outcome likelihood.

## P1.3. Conditional Probabilities and Posterior (1 points)

1. Derive:
   - $p(s_1, s_2 \mid t, y)$: the posterior distribution of player skills given the outcome.
   - $p(t \mid s_1, s_2, y)$: the conditional distribution of the outcome (Truncated Gaussian).
   - $p(y = 1)$: the probability of Player 1 winning.
2. Implement a Python script to compute these distributions for a given set of hyperparameters and outcomes, and validate your derivations numerically.

## P1.4. Gibbs Sampling (1 points)

1. Implement a Gibbs sampler for $p(s_1, s_2 \mid y)$ to estimate player skills:
   - Plot the samples and identify an appropriate burn-in period.
   - Compare the histogram of samples with the Gaussian posterior approximation.
2. Report computational trade-offs:
   - Evaluate accuracy vs. time using different numbers of samples post-burn-in.
   - Justify the optimal number of samples.

## P1.5

Assumed Density Filtering (ADF) is an online Bayesian inference method where the posterior distribution of one match is used as the prior for the next match. This approach is particularly useful for processing a sequence of data in real-time or iterative updates.

1. Use ADF with Gibbs sampling to process the matches in the `SerieA.csv` dataset:
   - Skip matches that ended in draws and update the skill distributions after each non-draw match.
   - Rank teams based on their final skill estimates and interpret the variance of the skill distributions.

2. Shuffle the order of matches randomly and re-run ADF. Does the result change? Why or why not?

## P1.6. One-Step-Ahead Predictions (1 points)

One-step-ahead predictions involve using the model to predict the outcome of each match based on the skills updated from prior matches. This is done iteratively for the entire dataset.

1. Implement a prediction function to decide whether Player 1 or Player 2 will win:
   - Use the current skill distributions to compute the probability $p(y = 1)$.
   - Return $+1$ if Player 1 is predicted to win and $-1$ otherwise.
2. Compute one-step-ahead predictions for all matches in the `SerieA.csv` dataset:
   - Calculate the prediction rate:

$$r = \frac{\text{Number of correct predictions}}{\text{Total number of matches}}$$

   - Compare your model's prediction rate to random guessing.

## P1.7. Message Passing Algorithm (1 points)

1. Implement a message-passing algorithm on the factor graph:
   - Use moment matching to approximate truncated Gaussians.
   - Compute the posterior skill distributions after one match.
2. Compare the message-passing results with Gibbs sampling:
   - Overlay the results (histograms and Gaussian approximations) in a single plot.

## P1.8. Custom Dataset (1 points)

1. Test your TrueSkill methods on a dataset of your choice. Possible datasets include:
   - Team sports (e.g., basketball, rugby).
   - Two-player sports (e.g., tennis, chess).
   - Online game data (e.g., multiplayer rankings).
2. Describe the source of your dataset and any preprocessing steps.
3. Report your findings, including a ranking of players or teams.

## P1.9. Model Extensions (2 points)

1. During your implementation of TrueSkill, you may have observed certain limitations in both the model and the inference algorithms. In this part of the project, you are tasked with defining and implementing your own extension of the model or method. Your extension should address a specific limitation or enhance the applicability of the model. Below are some suggestions:
   - Modify the model to account for draws or incorporate score differences in matches.
   - Introduce an a-priori advantage for certain players (e.g., white in chess).
   - Integrate external data or additional datasets to improve prediction accuracy.
   - Refine the representation of match outcomes by using detailed match scores or performance metrics.
   - Develop an improved prediction function that leverages additional insights from the data.

   Implement at least one extension to the model and test its performance. Analyze whether your extension results in:
   - Improved skill estimates.
   - Better prediction accuracy.
   - Reduced computational complexity or increased scalability.

   Discuss and justify your choice of extension, and provide insights on its impact.
2. Evaluate your extension on both the `SerieA.csv` dataset and your custom dataset:
   - Compare the results of your extended model with those from the baseline TrueSkill model.
   - Highlight any improvements in player rankings, prediction accuracy, or computational efficiency.
   - Provide a summary of your findings, supported by relevant plots or tables.

## Deliverables

- **Report:** Include derivations, plots, and explanations for each question. Be concise but thorough.
- **Code:** Provide a well-documented Python implementation with a README.
- **Results:** Include player rankings, prediction rates, and insights from model extensions.

## P2. QWOP (5 points)

### Description

In 2010, the QWOP game gained popularity for its challenging gameplay. Your task is to write a computer program to control the QWOP avatar by simulating its thigh and knee angles to maximize the distance it runs in the 100-meter event at the Olympic Games.
The QWOP physics engine has been implemented in Python and is available on the course website (P2.ipynb). The provided function accepts a list of 40 floating-point numbers (corresponding to 20 instructions for the angle of the thighs and 20 for the angle of the knees) as input and outputs the final $x$-position of the avatar's head. Your goal is to find an input vector that maximizes this output.

#### Simulation Setup

Ensure that you can call the Python function `sim(plan)` with a `plan` of 40 floating-point numbers in the range $[-1, 1]$. Use the provided visualization tool to observe the avatar's movements and understand how the input angles affect its motion.

### P2.1 Evaluation (5 points)

Optimize QWOP to maximize the distance $d$ the avatar travels. Design your own optimization methods to achieve this goal; do not use pre-existing online optimization code. Experiment with at least two of the following techniques:

- Markov Chain Monte Carlo (MCMC),
- Simulated annealing,
- Variants of gradient-based optimization,
- Evolutionary algorithms,
- Any other methods of your choice.

Evaluate the effectiveness of your methods:

- Report the best $x$-distance achieved by the avatar.
- Provide the length-40 input plan that resulted in the best performance.
- Compare and discuss the relative strengths and weaknesses of the methods you tried.

### Deliverables

- Submit your Python code, clearly labeled with the optimization methods implemented.
- Include a discussion of the best distance achieved, the length-40 input plan.

# P3: Generative Models

In this mini project, you will study and implement modern *generative models*. Generative models aim to learn the underlying probability distribution of data and generate new samples that resemble the observed data. Such models are widely used in image generation, data augmentation, and representation learning.
The project consists of three main parts:

- Variational Autoencoders (VAE)
- Score-based diffusion models on a low-dimensional distribution
- Latent Diffusion models for image generation

All implementations should be done using **PyTorch**. You are **not** required to implement your own neural network or optimizer from scratch. You are provided with Python templates for each part of the project. These templates contain only the necessary structure; you are expected to complete the missing parts.

## Background: Generative Models

Given data samples $\{\boldsymbol{x}_i\}_{i=1}^N \sim p_{\text{data}}(\boldsymbol{x})$, the goal of a generative model is to learn a distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ that approximates the true data distribution. Once learned, the model can be used to generate new samples by sampling from $p_{\boldsymbol{\theta}}(\boldsymbol{x})$.
Two important classes of generative models studied in this course are:

- **Latent variable models**, such as Variational Autoencoders (VAEs)
- **Score-based models**, also known as diffusion models

## P3.1 — Variational Autoencoder (VAE) (6 points)

A Variational Autoencoder introduces a latent variable $\boldsymbol{z}$ and models the data distribution as

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p(\boldsymbol{z})\, d\boldsymbol{z},$$

where $p(\boldsymbol{z})$ is typically a standard normal distribution.
Since the true posterior $p(\boldsymbol{z} \mid \boldsymbol{x})$ is intractable, VAEs use a variational approximation $q_\phi(\boldsymbol{z} \mid \boldsymbol{x})$ and optimize the *Evidence Lower Bound (ELBO).*
You should use the following Python template:

P3.1_VAE_template.py

### P3.1.1 Model (1 points)

Design a simple MLP encoder/decoder VAE:

- **Encoder** $q_\phi(z \mid x)$ maps an image $x \in \mathbb{R}^{784}$ to parameters of a Gaussian:

$$q_\phi(z \mid x) = \mathcal{N}\Big(z;\, \mu_\phi(\boldsymbol{x}), \text{diag}(\sigma_\phi(\boldsymbol{x})^2)\Big).$$

- **Prior** $p(z) = \mathcal{N}(0, \boldsymbol{I})$.
- **Decoder** $p_{\boldsymbol{\theta}}(x \mid z)$ maps $z \in \mathbb{R}^{d_z}$ back to the image domain.

**Choice of likelihood.** You can pick either:

- Bernoulli decoder: $p_{\boldsymbol{\theta}}(x \mid z) = \text{Bernoulli}(\hat{x}_{\boldsymbol{\theta}}(z))$ (use sigmoid at output), or
- Gaussian decoder: $p_{\boldsymbol{\theta}}(x \mid z) = \mathcal{N}(x;\, \hat{x}_{\boldsymbol{\theta}}(z), \sigma^2 I)$ (fixed $\sigma$).

**Deliverables.** Implement the VAE in python, Describe your architecture (layer sizes, activations, latent dimension $d_z$) and justify your choices.

### P3.1.2 ELBO derivation (2 points)

Explain (in your own words) the roles of:

- the latent variable $z$,
- the encoder (approximate posterior) $q_\phi(z \mid x)$,
- the decoder (likelihood) $p_{\boldsymbol{\theta}}(x \mid z)$,
- and the prior $p(z)$.

Starting from $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$, derive the evidence lower bound (ELBO):

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \;\geq\; \mathbb{E}_{q_\phi(z|x)}[\log p_{\boldsymbol{\theta}}(x \mid z)] - \mathrm{KL}(q_\phi(z \mid x) \,\|\, p(z)).$$

**Explain what each term does:**

- the reconstruction term $\mathbb{E}[\log p_{\boldsymbol{\theta}}(x \mid z)]$,
- the KL term $\mathrm{KL}(q_\phi(z \mid x) \| p(z))$.

Write the final objective you minimize in practice (negative ELBO).

### P3.1.3 Train the VAE (2 points)

Train the VAE on MNIST.

**Hint (reparameterization trick).** To backpropagate through sampling, use:

$$z = \mu_\phi(\boldsymbol{x}) + \sigma_\phi(\boldsymbol{x}) \odot \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \boldsymbol{I}).$$

- Show training curves (ELBO or negative ELBO, and optionally reconstruction vs KL).
- State your hyperparameters (learning rate, batch size, number of epochs, $d_z$).

### P3.1.4 How generation works in a VAE (1 points)

Explain how to generate new images:

1. Sample $z \sim p(z) = \mathcal{N}(0, \boldsymbol{I})$.
2. Compute decoder output $\hat{x} = \hat{x}_{\boldsymbol{\theta}}(z)$.
3. Convert $\hat{x}$ to an image (reshape to $28 \times 28$).

**Deliverables.** Plot a grid of generated images (at least $5 \times 5$). Briefly comment on quality and common failure cases.

## P3.2 — Score-Based Diffusion Model on a Spiral Distribution

Score-based models learn the *score function*

$$\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}),$$

which points toward regions of high probability. These models are trained using a *diffusion process* that gradually adds noise to data, followed by a learned reverse process that removes noise.
You should use the following Python template:

P3.2_Score_Diffusion_template.py

**Data Distribution** In this part, the data distribution is a **2D spiral Gaussian mixture**. The spiral distribution is given and must be used.

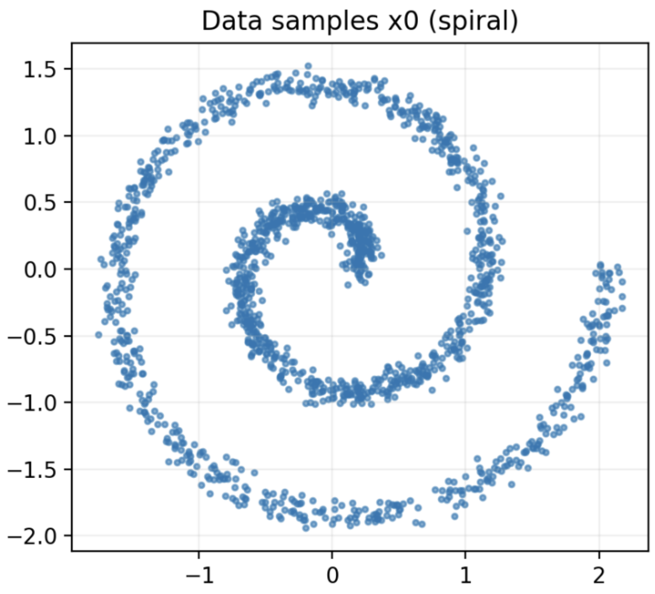

*Figure 1:* Spiral dataset $\boldsymbol{x}_0 \sim p_0(\boldsymbol{x})$.

We will use a spiral curve parameterized by $\tau$:

$$r(\tau) = a + b\tau, \qquad \mu(\tau) = \begin{bmatrix} r(\tau)\cos\tau \\ r(\tau)\sin\tau \end{bmatrix}.$$

A simple way to define a probability distribution on the spiral is a uniform mixture of isotropic Gaussians whose means are points on the spiral:

$$p_0(\boldsymbol{x}) = \frac{1}{M}\sum_{m=1}^{M} \mathcal{N}(\boldsymbol{x} \mid \mu(\tau_m), \sigma_0^2 \boldsymbol{I}).$$

You may assume that starter code is provided to generate samples from $p_0$.

### P3.2.1 Forward SDE and discretization (2 points)

**Idea (what is an SDE here?).** A **stochastic differential equation (SDE)** describes how a random variable evolves over continuous time. In diffusion models, we use an SDE to **gradually add noise** so that the data distribution moves from $p_0$ to a simple distribution (close to Gaussian).

**Forward (variance-preserving) SDE.** Let $\boldsymbol{x}(t) \in \mathbb{R}^2$ for $t \in [0,1]$. We define the forward (noising) process on $t \in [0,1]$ by the variance-preserving (VP) SDE:

$$d\boldsymbol{x}(t) = -\frac{1}{2}\beta(t)\boldsymbol{x}(t)\,dt + \sqrt{\beta(t)}\,d\boldsymbol{w}(t),$$

where $\boldsymbol{w}(t)$ is standard Brownian motion and $d\boldsymbol{w}(t)$ is a random increment with $d\boldsymbol{w}(t) \sim \mathcal{N}(0, dt)$. $\beta(t)$ is a **noise schedule**. We use a linear schedule:

$$\beta(t) = \beta_0 + (\beta_1 - \beta_0)t, \qquad \beta_0 = 0.001,\ \beta_1 = 0.2.$$

To implement the process we discretize time with steps $k = 0, 1, \dots, K$ and $\Delta t = 1/K$. Using Euler–Maruyama, the forward update can be written as

$$\boxed{\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \tfrac{1}{2}\beta_k\boldsymbol{x}_k\,\Delta t + \sqrt{\beta_k\Delta t}\,\boldsymbol{\xi}_k, \quad \boldsymbol{\xi}_k \sim \mathcal{N}(0, \boldsymbol{I}), \quad \beta_k = \beta(k/K).} \tag{0.1}$$

This is the forward process that maps clean data $\boldsymbol{x}_0$ to noisy samples $x_1, \dots, \boldsymbol{x}_k$.

**A useful closed form (for the marginal).** Define

$$\alpha_k = \exp\Big(-\beta(t_{k-1})\Delta t\Big), \qquad \bar{\alpha}_k = \prod_{j=1}^{k}\alpha_j.$$

Then the forward process implies the conditional distribution

$$q(\boldsymbol{x}_k \mid \boldsymbol{x}_0) = \mathcal{N}\Big(\boldsymbol{x}_k; \sqrt{\bar{\alpha}_k}\,\boldsymbol{x}_0,\ (1-\bar{\alpha}_k)\boldsymbol{I}_2\Big).$$

**Tasks.**

- Derive $p_k(\boldsymbol{x})$ the Distribution of $\boldsymbol{x}_k$ given that $\boldsymbol{x}_0 \sim p_0$ (a spiral Gaussian mixture).
- Run the forward process (0.1) starting from $\boldsymbol{x}_0 \sim p_0$. Plot the samples for at least 5 different steps $k$ (including $k = 0$ and $k = K$).
- Using the expression for $p_k(\boldsymbol{x})$ above, explain what you observe at the last step $\boldsymbol{x}_k$. In particular: why does $\boldsymbol{x}_k$ look close to a Gaussian when $\bar{\alpha}_K$ is very small?

### P3.2.2 Reverse process using the true score (2 points)

The forward process destroys structure by adding noise. The goal of the reverse-time process is to **start from the simple distribution** (near $p_K$) and **move back toward the data distribution** $p_0$, so we can *generate new samples.*
The reverse-time SDE corresponding to the VP forward SDE can be written as:

$$d\boldsymbol{x}(t) = \Big( -\tfrac{1}{2}\beta(t)\boldsymbol{x}(t) - \beta(t)\nabla_{\boldsymbol{x}} \log p_t(\boldsymbol{x}(t))\Big)\, dt + \sqrt{\beta(t)}\, d\bar{W}(t),$$

where $p_t$ is the distribution of $\boldsymbol{x}(t)$. As can be seen for the VP SDE, the reverse-time dynamics depend on the *score*

$$s_k(\boldsymbol{x}) := \nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x}).$$

In this toy problem you *do* have access to $p_k(\boldsymbol{x})$, so you can compute the true score.

**Tasks.**

- Derive the score $\nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x})$ for the spiral distribution.
- Implement the reverse-time discretizations (Euler) that uses the true score and starts from samples $\boldsymbol{x}_k \sim \mathcal{N}(0, \boldsymbol{I})$.
- Plot the distributions along the reverse trajectory.
- Compare the final generated samples (after reaching $k = 0$) to the original clean spiral samples.
- Explain how this reverse process can be used for generation.
- Explain why the score term is essential (what happens if you remove it?).

### P3.2.3 Learning the score (4 points)

**Why learn the score?** In real problems (e.g., images), the true density $p_k(\boldsymbol{x})$ is unknown, so we cannot compute $\nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x})$ directly. Instead, we learn a neural network $s_{\boldsymbol{\theta}}(\boldsymbol{x}, k)$ that approximates the score.

**Objective (score matching).** A common alternative objective is score matching, which avoids needing $p_k(\boldsymbol{x})$ explicitly. the **score matching** objective (Hyvärinen):

$$\mathcal{J}(\theta) = \mathbb{E}_{k\sim \mathrm{Unif}\{1,\ldots,K\},\, \boldsymbol{x}_k\sim p_k}\Big[\tfrac{1}{2}\|s_{\boldsymbol{\theta}}(\boldsymbol{x}_k, t_k)\|^2 + \underbrace{\nabla_{\boldsymbol{x}} \cdot s_{\boldsymbol{\theta}}(\boldsymbol{x}_k, t_k)}_{\text{divergence}}\Big], \tag{0.2}$$

where $\nabla_{\boldsymbol{x}} \cdot s_{\boldsymbol{\theta}}$ is the divergence (the trace of the Jacobian w.r.t. $x$). In the optimal point $\theta^*$, the objective function (0.2) enforces:

$$s_{\theta^*}(\boldsymbol{x}_k, t_k) = \nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x}) \tag{0.3}$$

So by minimizing (0.2), $s_{\boldsymbol{\theta}^*}(\boldsymbol{x}_k, t_k)$ is the good approximation of $\nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x})$.
**Implementation note.** In the Python helper code, the implementation of the divergence term is provided.
**Tasks.**

- Design your score network. Explain how you include time $t$ (e.g. concatenate $t$ to $x$).
- Train a neural network $s_{\boldsymbol{\theta}}(\boldsymbol{x}, k)$ using the divergence-based score matching objective (0.2) (using the provided Hutchinson estimator code).
- Report training details and show a training curve.
- On a 2D grid, visualize the learned vector field $s_{\boldsymbol{\theta}}(\boldsymbol{x}, t)$ for some noise levels $t$.

### P3.2.4 Generation with the learned score (2 points)

Now repeat Task 2.2, but replace the true score $s_k(\boldsymbol{x}) = \nabla_{\boldsymbol{x}} \log p_k(\boldsymbol{x})$ with your learned score $s_{\boldsymbol{\theta}}(\boldsymbol{x}, t_k)$ in (0.2).
**Tasks.**

- Plot the generated samples and compare them visually to:
  1. the clean spiral data, and
  2. the samples generated using the *true* score from P3.2.2
- Briefly discuss: what is the main gap between the learned-score generation and the true-score generation?

You should use the following Python template:

P3_Q2_spiral_score_template.py

**Reference** The Hutchinson trace estimator is described in:

- Hutchinson, M. F., *A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines*, Communications in Statistics (1989).

## P3.3 — Latent Diffusion Models

Applying diffusion models directly to high-dimensional images is computationally expensive. A common solution is to apply diffusion in a *low-dimensional latent space* learned by a VAE. This idea is known as a **latent diffusion model**.
You should use the following Python template:

P3_Q3_Latent_Diffusion_template.py

### P3.3.1 Why diffusion on pixels is hard (1 points)

Explain why directly applying the P3.2 objective and sampling procedure to image pixels is challenging.

### P3.3.2 Map MNIST to a low-dimensional latent space (1 points)

Using your trained VAE from P3.1:

- For each MNIST image $\boldsymbol{x}$, compute a latent representation. A simple choice is $\boldsymbol{z} = \mu_{\boldsymbol{\phi}}(\boldsymbol{x})$ (the encoder mean).
- Collect a dataset of latent codes $\{\boldsymbol{z}_i\}$.
- Report the latent dimension $d_z$ and visualize the latent dataset for $d_z = 2$ (scatter plot).

### P3.3.3 Learn a score model in latent space (2 points)

Now repeat P3.2 (forward, score learning, reverse) but on latent vectors $\boldsymbol{z} \in \mathbb{R}^{d_z}$.

- The forward process adds Gaussian noise in latent space.
- Train a score network $s_{\boldsymbol{\theta}}(\boldsymbol{z}_k, t_k)$ using the divergence-based (Hyvärinen) score matching objective..
- Show training curves.

### P3.3.4: Decode generated latents back to images (1 points)

Generate latent samples $\tilde{\boldsymbol{z}}$ by running the reverse process in latent space. Then generate images by passing $\tilde{\boldsymbol{z}}$ through the **decoder**:

$$\tilde{\boldsymbol{x}} = \text{Decoder}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{z}}).$$

- Plot a grid of generated images (at least $5 \times 5$).

## P3.3.5: Compare Part 1 vs Part 3 (1 points)

Visually compare:

- samples from the VAE alone (P3.1), vs
- samples from latent diffusion (P3.3).
- Which one looks better? Why do you think that happens?
- What kinds of mistakes do you see in each method?

**Reference** The Latent Diffusion models is described in:

- R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, *High-Resolution Image Synthesis with Latent Diffusion Models*, 2022.

## P3.4 —Conditional generation (3 points)

Make generation **conditional** on the digit label $y \in \{0, \ldots, 9\}$. For example, train a conditional score model $s_\theta(z, k, y)$ (or an equivalent conditioning method), and show that you can generate chosen digits on demand.

## Deliverables

- **Report:** Include derivations, plots, and explanations for each question. Be concise but thorough.
- **Code:** Provide a well-documented Python implementation with a README.
- **Results:** Include player rankings, prediction rates, and insights from model extensions.

# Bibliography

[1] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, *High-Resolution Image Synthesis with Latent Diffusion Models*, 2022.

[2] M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics – Simulation and Computation*, 18(3):1059–1076, 1989.