Universidad Autónoma de Madrid

Escuela Politécnica Superior

24|25

Bachelor thesis

Generative Artificial Intelligence through Score-Based Continuous-Time Diffusion Models

Gabriel López-Asiaín López

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as

BACHELOR THESIS

Generative Artificial Intelligence through Score-Based Continuous-Time Diffusion Models

Author: Gabriel López-Asiaín López
Advisor: Alberto Suárez González

May 2025

Gabriel López-Asiaín López
Generative Artificial Intelligence through Score-Based Continuous-Time Diffusion Models

Gabriel López-Asiaín López
C\ Francisco Tomás y Valiente Nº 11

PRINTED IN SPAIN

Agradecimientos

En primer lugar, me gustaría expresar mi agradecimiento a todos los profesores que me han acompañado a lo largo de la carrera. En especial, quiero dar las gracias a mi tutor, Alberto Suárez, por su implicación, su paciencia y el esfuerzo constante que ha dedicado para ayudarme a comprender los conceptos necesarios para poder llevar a cabo este proyecto.

También quiero dar las gracias a mis amigos de clase por su constante ayuda incondicional. Quiero agradecer especialmente a Fernando por haber sido un compañero de prácticas que no creo que me haya merecido y a Luis por haberme aguantado durante todo un año en nuestras aventuras por Australia.

Y sobre todo, muchas gracias a mi familia.

Resumen

El desarrollo de inteligencia artificial generativa representa un punto de inflexión en el aprendizaje automático moderno, permitiendo la creación de datos nuevos y realistas a partir de grandes conjuntos de datos. Estas técnicas, basadas en la estimación de distribuciones de datos, han revolucionado áreas como la visión por computadora y el procesamiento del lenguaje natural. Entre los marcos generativos más destacados se encuentran los autocodificadores variacionales (VAEs), las redes generativas adversarias (GANs) y, más recientemente, los modelos de difusión, que ofrecen un equilibrio entre calidad de generación, diversidad muestral y estabilidad de entrenamiento.

En este contexto, los modelos de difusión o modelos basados en funciones score proponen un nuevo marco teórico para la generación de datos, basado en la estimación del gradiente del logaritmo de la función de densidad de probabilidad (la función de score) de datos perturbados por ruido. A partir de dicha estimación, se resuelve una ecuación diferencial estocástica (SDE) inversa en el tiempo para lograr transformar muestras de una distribución de ruido en datos realistas. Este enfoque generaliza enfoques anteriores como los autocodificadores de eliminación de ruido, proporcionando una serie de herramientas que permiten el cálculo exacto de la verosimilitud y la generación sujeta a restricciones, manteniendo un alto rendimiento y calidad.

Este trabajo presenta un estudio completo e implementación de un modelo generativo basado en funciones score, con énfasis en la modularidad y el rigor teórico. Se ha desarrollado una implementación en Python desde cero, estructurada en varios módulos para la estimación del score y la generación de muestras mediante la resolución de SDEs. Además, la implementación permite la generación de imágenes condicionadas a una clase con la ayuda de clasificadores y cuenta con herramientas adicionales para garantizar la reproducibilidad y facilitar la experimentación.

El modelo ha sido evaluado empíricamente utilizando los conjuntos de imágenes MNIST y CIFAR-10. Los experimentos realizados cubren distintas configuraciones de red y parámetros del modelo. El rendimiento se ha evaluado tanto cuantitativamente como cualitativamente. Los resultados obtenidos ilustran cómo el modelo es capaz de capturar con precisión la distribución subyacente de los datos y generar muestras diversas y de calidad.

Palabras clave

Inteligencia Artificial Generativa, Modelos Generativos Basados en Score, Ecuaciones Diferenciales Estocásticas, Aprendizaje Profundo, Visión por Computador, Python

Abstract

The development of generative artificial intelligence represents a turning point in modern machine learning, enabling the creation of new and realistic data from large datasets. These techniques, based on the estimation of data distributions, have revolutionized fields such as computer vision and natural language processing. Among the most prominent generative frameworks are variational autoencoders (VAEs), generative adversarial networks (GANs), and, more recently, diffusion models, which offer a balance between generation quality, sample diversity, and training stability.

In this context, diffusion models or score-based models propose a new theoretical framework for data generation, based on estimating the gradient of the logarithm of the probability density function (the score function) of data perturbed by noise. From this estimation, a time-reversed stochastic differential equation (SDE) is solved to transform samples from a noise distribution into realistic data. This approach generalizes previous frameworks such as denoising autoencoders, providing a set of tools that enable exact likelihood computation and sampling under conditioning constraints, maintaining high performance and generation quality.

This work presents a comprehensive study and implementation of a score-based generative model, with an emphasis on modularity and theoretical rigor. A Python implementation has been developed from scratch, structured into differentiated modules for score estimation and sample generation through SDE solving. Furthermore, the implementation supports the generation of images conditioned to a class through classifier guidance and includes additional tools to ensure reproducibility and facilitate experimentation.

The model has been empirically evaluated on the MNIST and CIFAR-10 image datasets. The experiments conducted cover various network configurations and model parameters. Performance has been evaluated both quantitatively and qualitatively. The results illustrate that the model is capable of accurately capturing the underlying data distribution and generating both high-quality and diverse samples.

Keywords

Generative Artificial Intelligence, Score-based generative models, Stochastic Differential Equations, Deep Learning, Computer Vision, Python

Table of Contents

Lists

List of equations

List of figures

List of tables

1

Introduction

Over the past decade, generative artificial intelligence (AI) has emerged as one of the most innovative fields in machine learning, enabling the creation of highly realistic and complex data across a wide range of modalities. Generative models are systems designed to learn a data distribution in such a way that they can sample new, plausible instances from it. This capability not only allows for the generation of synthetic data but also supports tasks like data reconstruction [1], style transfer [2], and solving inverse problems [3].

The development of generative models has been historically grounded in two main paradigms: likelihood-based models and implicit models. Likelihood-based approaches, such as autoregressive models [4], normalizing flows [5], and variational autoencoders (VAEs) [6], offer strong theoretical guarantees and interpretable likelihood estimation, but often require restrictive model architectures or computationally intensive approximations. In contrast, implicit models, and most notably generative adversarial networks (GANs) [7], allow for highly expressive generation through adversarial training, though they tend to suffer from issues like mode collapse and unstable optimization dynamics.

To address these challenges, a new class of models known as score-based generative models [8] (SGMs) has recently attracted considerable interest. These models operate by learning the gradient of the log-density, known as the score function, rather than modeling the density directly. This change removes the necessity of computing normalization constants while still maintaining a high degree of expressiveness and sample quality. Initially paired with Markov Chain Monte Carlo (MCMC) methods for sampling, early score-based methods encountered limitations in regions of low data density [9], motivating the use of multiple noise scales.

A significant advance was achieved by interpreting these multiscale perturbations as discretizations of stochastic differential equations (SDEs). This interpretation led to the formulation of score-based generative modeling with SDEs [10], a framework that generalizes earlier approaches and unifies them under a continuous-time generative process. By training a time-dependent neural network to approximate the score of intermediate noisy distributions, one can simulate a reverse-time SDE that transforms simple noise into complex data. Furthermore, this framework supports advanced sampling techniques, such as predictor-corrector and ODE-based methods, and allows for likelihood computation

and constrained sample generation.

1.1 Goals and scope

This thesis explores the theoretical foundations and practical implementation of score-based generative modeling techniques using stochastic differential equations (SDEs). The objective is to provide a clear and rigorous overview of the mathematical and algorithmic principles underlying these models, and to develop a modular and well-documented software that illustrates the functioning of score-based generative models and allows for further experimentation.

To that end, the project involves a comprehensive study of score matching, noise perturbation schemes, and the role of SDEs in defining continuous-time generative processes. From this basis, we implement from scratch the core components of the generative pipeline, including the neural network used to estimate the score function, SDE solvers, and training routines, focusing on clarity, modularity, and reproducibility. Finally, we aim to illustrate the functioning of the system by testing the performance on benchmark datasets.

1.2 Structure of the document

This thesis is divided into five chapters, the first one being the introduction. Chapter 2 presents the theoretical background, covering score estimation techniques, methods for reversing stochastic differential equations, log-likelihood computation, and evaluation metrics for generative models. Chapter 3 outlines the software development process, including requirements analysis, system design, and implementation methodology. Chapter 4 reports the experimental results obtained on different datasets. Finally, Chapter 5 summarizes the main conclusions of the project and outlines possible directions for future work.

2

State Of The Art

This chapter provides an overview of the theoretical foundations relevant to score-based generative modeling. Section 2.1 introduces the core ideas behind score functions and their estimation, and outlines early sampling techniques. Section 2.2 describes the transition to continuous-time generative models, formalizing the use of stochastic differential equations and presenting methods for sampling, controlled generation, and exact likelihood computation. Section 2.3 introduces standard evaluation metrics for generative models, focusing on the Inception Score and the Fréchet Inception Distance, which are used to assess the quality and diversity of generated samples.

2.1 The score function, score-based models, and score matching

Given a dataset $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$, where each sample is drawn independently from an underlying distribution $p(\mathbf{x})$, the objective of generative modeling is to learn a model that closely approximates $p(\mathbf{x})$, enabling the generation of new samples that follow the same distribution.

The natural way to do this, as it is done in likelihood-based models, is to try to approximate directly the probability density function (p.d.f.) (or probability mass function) of the data distribution. Given any real-valued function $f_\theta(\mathbf{x})$, we can define a p.d.f. from it in the following way:

$$p_\theta(\mathbf{x}) = \frac{e^{-f_\theta(\mathbf{x})}}{Z_\theta} \tag{2.1}$$

where Z_θ is a normalizing constant depending on θ that ensures that $\int p_\theta(\mathbf{x})dx = 1$. This normalizing constant is often costly to compute, which makes this approach undesirable. An alternative to avoid the computation of the normalizing constant is to work with the score function, defined as the gradient of the log-likelihood:

$$\mathbf{s}(\mathbf{x}) = \nabla_x \log p(\mathbf{x}). \tag{2.2}$$

A model that approximates this function, denoted as $\mathbf{s}_\theta(\mathbf{x}) \approx \nabla_x \log p(\mathbf{x})$, is referred to as a *score-*

based model. It is important to note that the score function is unaffected by any constant multiplicative factor in the probability density function (p.d.f.), as such a factor cancels out when taking the logarithm and differentiation. Specifically, if we consider the energy-based model defined in 2.1 we observe that when computing its score, the normalizing constant vanishes:

$$\mathbf{s}_\theta(\mathbf{x}) = \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x}) = -\nabla_{\mathbf{x}} f_\theta(\mathbf{x}) - \underbrace{\nabla_{\mathbf{x}} \log Z_\theta}_{=0} = -\nabla_{\mathbf{x}} f_\theta(\mathbf{x}).$$

Similarly to likelihood-based models, score-based models can be trained by minimizing the Fisher divergence between the true score function and the model score function. The Fisher divergence is defined as:

$$\mathbb{E}_{p(\mathbf{x})}\left[\|\nabla_{\mathbf{x}} \log p(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x})\|_2^2\right] = \int p(\mathbf{x}) \|\nabla_{\mathbf{x}} \log p(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x})\|_2^2 \, d\mathbf{x}. \tag{2.3}$$

However, directly computing the Fisher divergence is not possible since it depends on the true score function, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$, which is unknown. Nevertheless, there is a family of methods known as *score matching* [11,12] that provide an alternative approach to estimating the Fisher divergence without requiring explicit knowledge of the true score function.

Note that the score matching objective does not require the model $\mathbf{s}_\theta(\mathbf{x})$ to be the actual score of some function, which gives more flexibility to this method. In fact, the only requirement is for the score model to be a vector-valued function with the same input and output dimensionality, which is easy to satisfy in practice.

2.1.1 Langevin Dynamics

Once we have trained a score-based model $\mathbf{s}_\theta(\mathbf{x}) \approx \nabla_{\mathbf{x}} \log p(\mathbf{x})$, we can use an iterative procedure called *Langevin dynamics* to draw samples from it. Langevin dynamics provides a Monte Carlo Markov Chain (MCMC) procedure to sample from a distribution $p(\mathbf{x})$ using only its score function $\nabla_{\mathbf{x}} \log p(\mathbf{x})$. Specifically, it initializes the chain from an arbitrary prior distribution $\mathbf{x}_0 \sim \pi(\mathbf{x})$, and then iterates according to the following rule:

$$\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \epsilon \nabla_{\mathbf{x}} \log p(\mathbf{x}) + \sqrt{2\epsilon}\mathbf{z}_i, \quad i = 0, 1, \ldots, K, \tag{2.4}$$

where $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$ and ϵ is the step size. When $\epsilon \to 0$ and $K \to \infty$, $\mathbf{x}_K$ obtained from the procedure in (2.4) converges to a sample from $p(\mathbf{x})$ under some regularity conditions. In practice, the error is negligible when ϵ is sufficiently small and K is sufficiently large.

Since it only accesses $p(\mathbf{x})$ through its score function, the score-model $\mathbf{s}_\theta(\mathbf{x})$ can be replaced by $\nabla_{\mathbf{x}} \log p(\mathbf{x})$ in equation (2.4) to generate samples from our score-based model.

Nonetheless, this approach has had limited success in practice. The main pitfall is the fact that the estimated score function is inaccurate in low density regions [9], as it does not have enough information due to the lack of data samples. This can be seen directly in equation (2.3). Since the Fisher divergence is weighted by the data distribution $p(\mathbf{x})$, these distances are not penalized in low density regions.

More specifically, as the data distribution is always estimated using i.i.d. samples $\{\mathbf{x}_i\}_{i=1}^N \overset{\text{i.i.d.}}{\sim} p(\mathbf{x})$, if we consider any region $\mathcal{R} \subset \mathbb{R}^d$ such that $p(\mathcal{R}) \approx 0$, in most cases, $\{\mathbf{x}_i\}_{i=1}^N \cap \mathcal{R} = \emptyset$, and score matching will not have sufficient data samples to estimate $\nabla_{\mathbf{x}} \log p(\mathbf{x})$ accurately for $\mathbf{x} \in \mathcal{R}$.

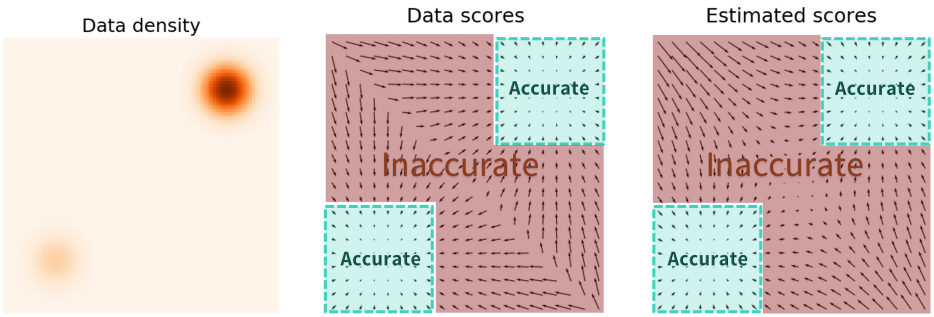


Figure 2.1: Simplified example of how score estimation is inaccurate on lower density regions (Source [8]).

2.1.2 Annealed Langevin dynamics

To address the limitations of Langevin dynamics, [8] proposed a new method called *annealed Langevin dynamics*. The idea is to perturb the data with noise, in order to populate the low-density regions with samples and then train the model on the noisy dataset.

Greater noise levels can cover more low-density regions, improving score estimation. However, excessive noise leads to over-corruption of the data, significantly altering its resemblance to the original distribution. On the other hand, lower noise levels do not significantly alter the data distribution but fail to sufficiently cover low-density regions as desired.

The idea is to use different noise levels during training, allowing the model to learn from the noisy data and gradually adapt to the original data distribution. We choose to perturb the data with isotropic Gaussian noise, which is simple and effective in practice, with a total of L noise levels with standard deviations $\sigma_1 < \sigma_2 < \cdots < \sigma_L$. Typically this sequence is chosen to be a geometric progression ensuring σ_1 is small enough and σ_L comparable to the maximum pairwise distance between all training data points [13]. By perturbing the distribution $p(\mathbf{x})$ with each of the Gaussian noise $\mathcal{N}(\mathbf{0}, \sigma_i^2 \mathbf{I})$, we obtain a sequence $\{p_{\sigma_i}(\mathbf{x})\}_{i=1}^L$ distributed as follows:

$$p_{\sigma_i}(\mathbf{x}) = \int p(\mathbf{y}) \mathcal{N}(\mathbf{x}; \mathbf{y}, \sigma_i^2 I)\, d\mathbf{y}. \tag{2.5}$$

Note that sampling from $p_{\sigma_i}(\mathbf{x})$ is equivalent to sampling from $\mathbf{x} \sim p(\mathbf{x})$ and computing $\mathbf{x} + \sigma_i \mathbf{z}$, with $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

The next step is to train a score-based model to estimate the score of the dataset perturbed with each of the noise scales $\mathbf{s}_\theta(\mathbf{x}, i) = \nabla_{\mathbf{x}} \log p_{\sigma_i}(\mathbf{x})$, using the score matching techniques already mentioned. This is called a *Noise Conditional Score-Based Model*, also called a *Noise Conditional Score Network*, or *NCSN*. The score matching objective for the NCSN would be the following:

$$\min_\theta \sum_{i=1}^{L} \lambda(i) \mathbb{E}_{p_{\sigma_i}(\mathbf{x})} \left[\| \nabla_{\mathbf{x}} \log p_{\sigma_i}(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x}, i) \|_2^2 \right].$$

Here $\lambda(i)$ is a weighting factor that can be used to balance the contribution of each noise level to the overall objective. In practice, it is common to use $\lambda(i) = \sigma_i^2$.

When working with images, the NCSN is usually modeled using a U-Net [14], a family of deep convolutional neural networks that has demonstrated strong performance in image processing and recognition tasks. The U-Net architecture combines a contracting path (downsampling) that captures contextual information with a symmetric expansive path (upsampling) that enables precise spatial localization, leveraging skip connections to preserve high-resolution features throughout the network. Details about the particular implementation of the U-Net for this purpose can be found in the following chapter.

Once the NCSN is trained, we can use annealed Langevin dynamics to generate samples from the original distribution $p(\mathbf{x})$. The idea is to start drawing a sample from $p_{\sigma_L}(\mathbf{x})$ which is typically a known distribution such as Gaussian noise, which we call *prior distribution*. Then, for each subsequent noise level σ_i, Langevin dynamics is applied using the final samples from the previous step as initialization, while gradually decreasing the step size according to $\alpha_i = \epsilon \cdot \sigma_i^2 / \sigma_1^2$. This process continues in reverse order, from L down to 1, until reaching $q_{\sigma_1}(\mathbf{x})$, which approximates the true data distribution $p(\mathbf{x})$ when $\sigma_1 \approx 0$.

2.2 Continuous-time generative models

Naturally, the more noise levels we use, the more accurate the samples generated by the model will be. By generalizing the number of noise scales to infinity, considering a continuous sequence of noise levels, we not only obtain better quality samples but also other advantages like exact log-likelihood computation or controllable generation. The idea is to perturb the data using a continuous-time stochastic process. This is a family of random variables $\{\mathbf{x}_t\}_{t \geq 0}$ indexed by time $t \geq 0$, such that $\mathbf{x}_t$ is a sample from the distribution $p_t(\mathbf{x})$ at time t.

A common and precise way to represent many continuous-time stochastic processes is through a stochastic differential equation (SDE), which describes the evolution of a stochastic process over time.

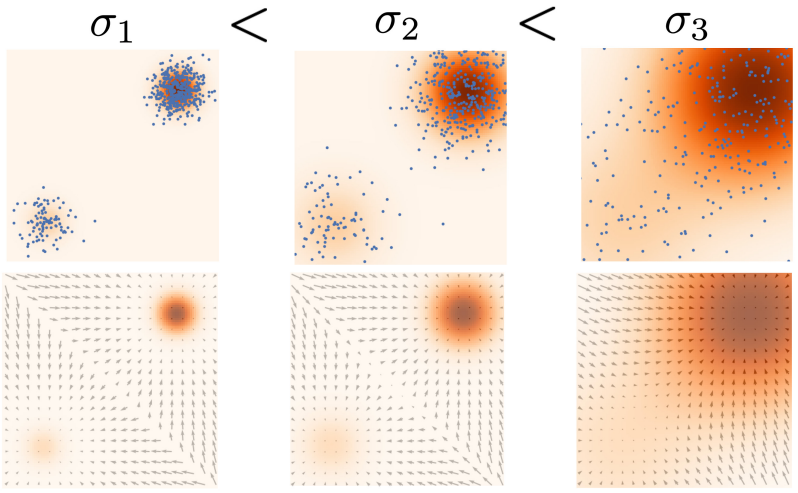


Figure 2.2: Illustration of how Annealed Langevin Dynamics uses progressively decreasing noise levels to draw a sample from the distribution. (Source [8]).

Specifically, the SDEs we will work with have the following form:

$$d\mathbf{x} = \mathbf{f}(\mathbf{x}, t)dt + g(t)d\mathbf{w}, \tag{2.6}$$

where $\mathbf{f}(\mathbf{x}, t) : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is a vector-valued function called the drift term, $\mathbf{g}(t)$ is a real-valued function called the diffusion term, dt represents a negative infinitesimal time step, and $\mathbf{w}$ is a standard Wiener process. The solution to an SDE is a stochastic process $\{\mathbf{x}_t\}_{t\geq 0}$ that evolves over time according to the drift and diffusion terms.

Let $p_t(\mathbf{x})$ denote the (marginal) probability density function of $\mathbf{x}(t)$. Here $t \in [0, T]$ is analogous to $i = 1, 2, \ldots, L$ when we had a finite number of noise scales, and $p_t(\mathbf{x})$ is analogous to $p_{\sigma_i}(\mathbf{x})$. Clearly, $p_0(\mathbf{x}) = p(\mathbf{x})$ is the data distribution since no perturbation is applied to data at $t = 0$. After perturbing $p(\mathbf{x})$ with the stochastic process for a sufficiently long time T, $p_T(\mathbf{x})$ becomes close to a tractable noise distribution $\pi(\mathbf{x})$, called a *prior distribution*.

A widely used stochastic process to perturb data is what is known as a Brownian motion with a time-dependent diffusion coefficient. When its variance is unbounded, we denote it by Variance Exploding (VE) diffusion process. It is defined by the following SDE:

$$d\mathbf{x} = g(t)d\mathbf{w}, \tag{2.7}$$

integrating from 0 to t we obtain the solution to the SDE:

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t g(s)d\mathbf{w}(s).$$

We can then compute the expected value and variance of $\mathbf{x}(t)$, which are given by:

$$\mathbb{E}[\mathbf{x}(t)] = \mathbf{x}(0), \quad \text{Var}[\mathbf{x}(t)] = \int_0^t g^2(s)ds.$$

With this, we can rewrite the solution to the SDE as:

$$\mathbf{x}(t) = \mathbf{x}(0) + \sqrt{\int_0^t g^2(s)ds} \cdot z, \quad z \sim \mathcal{N}(0,1). \tag{2.8}$$

For instance, it is common to choose a diffusion function that grows exponentially, with $g(t) = \sigma^t$. In this case, the accumulated variance is given by:

$$\int_0^t g^2(s)ds = \int_0^t \sigma^{2s}ds = \frac{1}{2\log\sigma}(\sigma^{2t} - 1).$$

And therefore the marginal distribution of $\mathbf{x}(t)$ given $x(0)$ is:

$$p_{0t}(\mathbf{x}(t) \mid \mathbf{x}(0)) = \mathcal{N}\left(\mathbf{x}(t); \mathbf{x}(0), \frac{1}{2\log\sigma}(\sigma^{2t} - 1)\mathbf{I}\right).$$

We call this the transition probability from $\mathbf{x}(0)$ to $\mathbf{x}(t)$. Typically we choose $p_{T=1}$ to be the prior distribution; in this particular example, when σ is large, the prior distribution is

$$\int p_0(\mathbf{y})\mathcal{N}\left(\mathbf{x}; \mathbf{y}, \frac{1}{2\log\sigma}(\sigma^2 - 1)\mathbf{I}\right)d\mathbf{y} \approx \mathcal{N}\left(\mathbf{x}; \mathbf{0}, \frac{1}{2\log\sigma}(\sigma^2 - 1)\mathbf{I}\right),$$

which is approximately independent of the data distribution and is easy to sample from.

Another common diffusion process used to perturb data is the Ornstein-Uhlenbeck process, which is defined by:

$$d\mathbf{x} = -\frac{1}{2}f(t)\,\mathbf{x}\,dt + g(t)\,d\mathbf{w}, \tag{2.9}$$

The diffusion function is typically chosen as $g(t) = \sqrt{f(t)}$, ensuring that the variance of the perturbed sample remains bounded (Variance Preserving).

2.2.1 Reversing the SDE for sample generation

Analogously to the annealed Langevin dynamics, we intend to sample from the prior distribution $p_T(\mathbf{x})$ and then reverse the perturbation process to obtain samples from the data distribution $p_0(\mathbf{x})$. For any given SDE, there exists a corresponding reverse SDE [15], which is given by:

$$d\mathbf{x} = \left[\mathbf{f}(\mathbf{x}, t) - g^2(t)\nabla_{\mathbf{x}} \log p_t(\mathbf{x})\right] dt + g(t)d\mathbf{w}. \tag{2.10}$$

Naturally, this reverse SDE is solved from T to 0. In order to do this, we first need to estimate the score function $\nabla_{\mathbf{x}} \log p_t(\mathbf{x})$ at each time step t. The new training objective for the score-based model is then:

$$\min_\theta \mathbb{E}_{t\sim\mathcal{U}(0,T)}[\lambda(t)\mathbb{E}_{\mathbf{x}(0)\sim p_0(\mathbf{x})}\mathbb{E}_{\mathbf{x}(t)\sim p_{0t}(\mathbf{x}(t)|\mathbf{x}(0))}[\|s_\theta(\mathbf{x}(t), t) - \nabla_{\mathbf{x}(t)} \log p_{0t}(\mathbf{x}(t) \mid \mathbf{x}(0))\|_2^2]], \tag{2.11}$$

where $\mathcal{U}(0,T)$ is a uniform distribution over $[0,T]$ and $\lambda(t) \in \mathbb{R}_{>0}$ denotes a positive weighting function typically chosen to be inversely proportional to $\mathbb{E}[\|\nabla_{\mathbf{x}} \log p_{0t}(\mathbf{x}(t) \mid \mathbf{x}(0))\|_2^2]$.

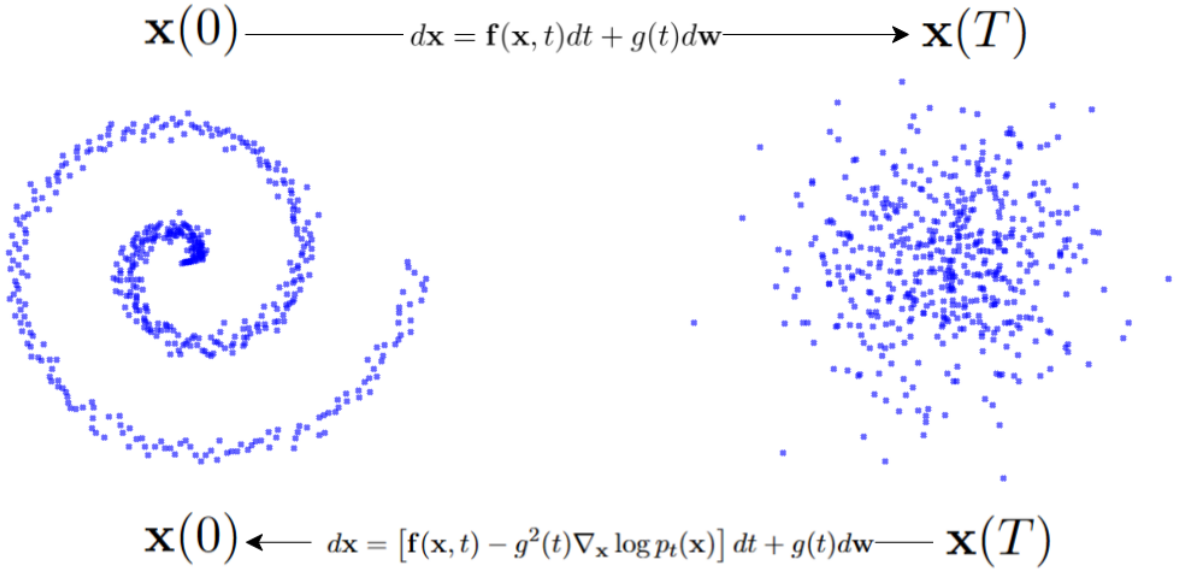


Figure 2.3: Illustration of the forward diffusion process transforming data into Gaussian noise, and the reverse process progressively denoising it to recover data samples.

SDE solvers

Now we can plug the estimated score function into the reverse SDE (2.10) and simulate a solution from T to 0 using a numerical solver. The simplest and most widely used method is the Euler-Maruyama method [16]. It is based on a simple discretization of the SDE, replacing dt with Δt and $d\mathbf{w}$ with $\mathbf{z}_t \sim \mathcal{N}(0, g^2(t)\Delta t\mathbf{I})$, obtaining the following iteration rule:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \left[\mathbf{f}(\mathbf{x}_t, t) - g^2(t)\nabla_{\mathbf{x}} \log p_t(\mathbf{x}_t)\right] \Delta t + g(t)\mathbf{z}_t. \tag{2.12}$$

Under some regularity conditions and sufficiently small step size Δt, the Euler-Maruyama method yields a consistent approximation to the solution of the reverse-time SDE. As $\Delta t \to 0$, the distribution of the generated samples converges to the true data distribution.

Predictor-Corrector methods

As we are only interested in sampling from each marginal distribution $p_t(\mathbf{x})$ separately, we can still apply MCMC methods to fine-tune the samples obtained by SDE solvers in each step. These mixed methods are called *Predictor-Corrector* samplers. The predictor can be any numerical SDE solver that predicts $\mathbf{x}(t + \Delta t) \sim p_{t+\Delta t}(\mathbf{x})$ from an existing sample $\mathbf{x}(t) \sim p_t(\mathbf{x})$ such as Euler-Maruyama. The corrector can be any MCMC procedure that solely relies on the score function, such as Langevin dynamics.

ODE solvers

Despite being able to generate high-quality samples, SDE solvers do not provide an exact method to compute log-likelihood. In [10] it is shown that it is possible to turn an SDE into an ordinary differential equation (ODE) such that the marginal distributions $\{p_t(\mathbf{x})\}_{t\in[0,T]}$ stay the same. The corresponding ODE of a SDE is called *probability flow ODE* and it is given by

$$d\mathbf{x} = \left[\mathbf{f}(\mathbf{x},t) - \frac{1}{2}g^2(t)\nabla_{\mathbf{x}}\log p_t(\mathbf{x})\right]dt. \tag{2.13}$$

With this, we can draw samples from the underlying data distribution by drawing a random sample from p_T, and integrating the ODE in the reverse time direction, from T to 0, to obtain a sample from p_0.

2.2.2 Controllable generation

Up to this moment, we have studied only how to draw unconditional samples from a data distribution. Now, it would be of interest to explore how to draw samples from the dataset conditioned to a class (e.g. generate a cat from a dataset of animals). In this case, we are interested in the distribution of $p(\mathbf{x}|y)$ where y is a given class label.

The main idea is to use classifiers that guide the diffusion process towards a certain class. We can train a classifier model $c_\phi(\mathbf{x}_t,t)$ on noisy images $\mathbf{x}_t$ to approximate $p(y|\mathbf{x}_t,t)$, and then use gradients $\nabla_{\mathbf{x}_t}c_\phi(\mathbf{x}_t)$ to guide the diffusion sampling process towards an arbitrary class label y.

There are specific methods that exploit the use of classifiers to improve generation (See [17]). However, there is a simpler approach, which is to use the Bayes rule to obtain the score of the conditioned distribution:

$$\nabla_{\mathbf{x}}\log p(\mathbf{x}|y) = \nabla_{\mathbf{x}}\log\left(\frac{p(y|\mathbf{x})p(\mathbf{x})}{p(y)}\right) = \nabla_{\mathbf{x}}\log p(\mathbf{x}) + \nabla_{\mathbf{x}}\log p(y|\mathbf{x}). \tag{2.14}$$

This simple rule allows us to sample from the conditioned distribution if we have an estimate of both the score and the gradient of $p(y|\mathbf{x}_t,t)$. In practice, we multiply $\nabla_{\mathbf{x}}\log p(y|\mathbf{x})$ in equation (2.14) by a parameter γ named gradient scale to control the influence of the classifier in the guidance process. The bigger this value is, the more biased the sample will be. Now, we only need to plug our score model $s_\theta(\mathbf{x},t)$ and the gradient of the log of the classifier $c_\phi(\mathbf{x},t)$ into equation (2.14) to obtain the conditioned score model:

$$s_y(\mathbf{x},t) = s_\theta(\mathbf{x},t) + \gamma\cdot\nabla_x c_\phi(\mathbf{x},t). \tag{2.15}$$

Then using $s_y(\mathbf{x},t)$ we can sample from the conditioned distribution using all the methods already described for unbiased sampling.

2.2.3 Exact log-likelihood computation

As we have already remarked, one of the advantages of continuous-time diffusion models is the possibility of computing the exact log-likelihood of the model. Given a sample $\mathbf{x}_0$ we want to compute its log-likelihood $\log p_0(\mathbf{x}_0)$. The ODE solver gives us a direct method to compute this quantity by using the probability flow ODE:

$$d\mathbf{x} = \left[\mathbf{f}(\mathbf{x}, t) - \frac{1}{2} g(t)^2 \nabla_{\mathbf{x}} \log p_t(\mathbf{x})\right] dt = \mathbf{h}(\mathbf{x}, t) dt.$$

If we integrate from from $t = 0$ to $t = T$, we get:

$$\mathbf{x}(T) = \mathbf{x}(0) + \int_0^T \mathbf{h}(\mathbf{x}, t) dt.$$

Now, applying the instantaneous change of variable formula from [14] to the equation $d\mathbf{x} = \mathbf{h}(\mathbf{x}, t)dt$ we obtain:

$$\frac{\partial \log p(\mathbf{x}(t))}{\partial t} = -tr\left(\frac{d\mathbf{h}}{d\mathbf{x}}(t)\right), \tag{2.16}$$

and again, integrating from 0 to T we get:

$$\log p(\mathbf{x}(0)) = \log p(\mathbf{x}(T)) + \int_0^T tr\left(\frac{d\mathbf{h}}{d\mathbf{x}}(t)\right),$$

so if we set $\mathbf{x}(0) = \mathbf{x}_0$, and draw a sample $\mathbf{x}_T$ from $p_{0t}(\mathbf{x}(T)|\mathbf{x}_0)$ so that $\mathbf{x}(T) = \mathbf{x}_T$ we end up with the following system to solve:

$$\begin{bmatrix} \mathbf{x}_T \\ \log p_0(\mathbf{x}_0) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0 \\ \log p_T(\mathbf{x}_T) \end{bmatrix} + \int_0^T \begin{bmatrix} \mathbf{h}(\mathbf{x}, t) \\ \operatorname{div} \mathbf{h}(\mathbf{x}, t) \end{bmatrix} dt, \tag{2.17}$$

where $\operatorname{div} \mathbf{h}(\mathbf{x}, t)$ is the divergence of the function h which is defined as the trace of the Jacobian. And therefore if $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ and $\mathbf{h}(\mathbf{x}, t) = (h_1(\mathbf{x}, t), h_2(\mathbf{x}, t), \ldots, h_d(\mathbf{x}, t))$ it can be computed as:

$$\operatorname{div} \mathbf{h}(\mathbf{x}, t) = \frac{\partial h_1}{\partial x_1}(\mathbf{x}, t) + \frac{\partial h_2}{\partial x_2}(\mathbf{x}, t) + \cdots + \frac{\partial h_d}{\partial x_d}(\mathbf{x}, t) \tag{2.18}$$

However, when d is big, which is the case when we are working with complex data such as images, computing this value directly is extremely costly in computational terms. A more efficient way to estimate the divergence of the score function rather than compute it directly is to use the Skilling-Hutchinson estimator [18, 19] of the divergence, which is an unbiased estimator of the divergence based on the following identity:

$$\operatorname{div} \mathbf{f}(\mathbf{x}) = \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})}[\boldsymbol{\epsilon}^\intercal J_{\mathbf{f}}(\mathbf{x}) \boldsymbol{\epsilon}]. \tag{2.19}$$

where $J_{\mathbf{f}}$ is the Jacobian matrix of f. This way we can draw k samples from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ and average

the quadratic form of the Jacobian of the score function to estimate the divergence of the score function.

The reason why this approach is much more efficient than computing the divergence directly is that computing the Jacobian-vector product $J_{\mathbf{f}}(\mathbf{x})\epsilon$ involves only a single pass in the reverse-mode automatic differentiation algorithm, and therefore if k is significantly smaller than d, the computational cost might be reduced as fewer derivatives are required to compute.

Now, as we intend to work with images, it would be interesting to understand this log-likelihood in terms of *bits per dimension*. This metric describes how many bits we would need to encode a particular example in our modeled distribution. Pixels can only take so many values, usually an integer between 0 and 255. Because of this, we need to discretize the distribution in order to compute the likelihood. If our distribution is $p_0(x)$ the re-scaled distribution is going to be $p_d(x) = \frac{1}{256^{C\cdot N}} p_0(x)$, where N is the number of pixels in the image and C is the number of color channels (usually 3 if the picture is RGB). With this we can then compute the bits/dim using the following formula:

$$\text{bits/dim}(x) := -\frac{\log_2 p_d(x)}{C \cdot N} = -\frac{\log_2 p_0(x) - C \cdot N \log_2(256)}{C \cdot N} = -\frac{\log p_0(x)}{C \cdot N \cdot \log(2)} + 8. \tag{2.20}$$

This is the metric we are going to use to evaluate the model. The value 8 in equation (2.20) is known as the color depth of the image and it represents the number of bits needed to encode a single pixel of a single channel.

2.3 Evaluation metrics for generative models

While log-likelihood provides a precise measure for evaluating and comparing probabilistic models, it does not directly assess the perceptual quality of individual samples generated by the model. In fact, evaluating the quality of generated images remains a challenging and often subjective task, as it involves aspects such as visual fidelity and diversity that are not fully captured by likelihood-based metrics. To address this, several quantitative metrics have been proposed in the literature, aiming to evaluate both the visual quality and the diversity of the generated data. Among these, two of the most widely used metrics for image evaluation are the *Inception Score* (IS) [20] and the *Fréchet Inception Distance* (FID) [21]. Both rely on the use of a pre-trained *Inception* network [22] to extract features from images, but differ in how they interpret these features.

2.3.1 Inception Score

The Inception Score (IS) aims to evaluate the quality of generated images by measuring two key aspects: (1) whether individual samples are recognizable and confidently classifiable, and (2) whether the set of generated samples is diverse across different semantic categories. Given a set of generated images $\{\mathbf{x}_i\}_{i=1}^N$, each image is passed through a pre-trained Inception network to obtain the conditional

label distribution $p(y|\mathbf{x}_i)$. The IS is then defined as:

$$\text{IS} = \exp\left(\mathbb{E}_{\mathbf{x}}\left[D_{\text{KL}}(p(y|\mathbf{x})\|p(y))\right]\right), \tag{2.21}$$

where D_{KL} denotes the Kullback–Leibler divergence [23], and $p(y) = \int p(y|\mathbf{x})p(\mathbf{x})d\mathbf{x}$ is the marginal distribution over labels estimated empirically. High-quality samples are expected to have a low-entropy conditional distribution $p(y|\mathbf{x})$ (i.e., be confidently classifiable), and the marginal $p(y)$ is expected to be high-entropy (i.e., the samples cover many classes). This way, a higher Inception Score indicates better visual quality and sample diversity.

Despite its popularity, IS has several limitations: it does not explicitly compare generated images to real ones, it is sensitive to the choice of the classifier, and it may not capture intra-class diversity.

2.3.2 Fréchet Inception Distance

The Fréchet Inception Distance (FID) provides a more direct comparison between the distribution of generated samples and the distribution of real images. Let $\mathcal{X}_r$ and $\mathcal{X}_g$ denote sets of real and generated images, respectively. Both sets are embedded into a high-level feature space using the activations of a pre-trained Inception network (typically from the penultimate layer). The empirical distributions of these features are then approximated by multivariate Gaussians with means and covariances (μ_r, Σ_r) and (μ_g, Σ_g), respectively. The FID is defined as:

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\right), \tag{2.22}$$

where Tr denotes the trace of a matrix. Lower FID scores indicate that the generated distribution is closer to the real one, both in terms of mean and covariance structure in the feature space.

FID is generally considered more robust than IS, as it directly measures the similarity between generated and real samples. It also correlates better with human judgment of visual quality. However, it assumes that the features follow a Gaussian distribution, and may be biased by the number of samples used to compute the statistics.

3

Software Development

The goal of this project is to implement a fully functional and well-documented system that allows for the application of the generative modeling methods previously introduced to synthesize new images from a given dataset. Rather than relying on existing high-level libraries or pre-trained models, this implementation has been developed from scratch with a strong emphasis on clarity and modularity.

In this chapter, we describe in detail the development process followed throughout the project. We begin by outlining the functional and non-functional requirements that shaped the design of the system; we then present the architectural and technical design choices made, including the structure of the software components and the interface exposed to the user. Finally, we describe the implementation process, including the methodology and the tools used to support development.

3.1 Requirements analysis

In this section, we present the analysis of the required functionalities of the score-based generative diffusion model. We will present the functional and nonfunctional requirements, as well as some use cases that will guide the development.

3.1.1 Functional requirements

We will first state the functional requirements that define the essential operations the system must support. Each requirement is specified to ensure the correct and complete functionality of the system.

FR-1 Given a dataset of images, the user must be able to obtain an estimate of the score function of the underlying data distribution:

- **FR-1.1** The software shall apply Gaussian noise to each input image at various noise levels using a selected SDE (VE or VP).
- **FR-1.2** The user must be able train a neural network on a selected dataset to predict the score of the perturbed images at each noise level using a denoising

score matching loss function.

FR-1.3 The model should be saved to disk for future use.

FR-2 The user must be able to generate unconditional samples from the trained model through the software. The user can optionally choose the sampling method among the following:

FR-2.1 Euler-Maruyama

FR-2.2 Predictor-Corrector

FR-2.3 ODE solver

FR-3 The user must be able to generate samples from the trained model conditioned on a given class label using the same methods as unconditional sampling. This must be done using a classifier trained on noisy images.

FR-4 The user must be able to compute the negative log-likelihood of images given the trained model, measured in bits per dimension.

3.1.2 Non-functional requirements

We now describe the non-functional requirements that the system must meet to ensure usability, compatibility, and maintainability.

NFR-1 The software must be compatible with Windows, Linux, and macOS operating systems.

NFR-2 The software must be implemented in Python 3.12.

NFR-3 The code must comply with the Google Python Style Guide [24].

NFR-4 The software must support execution on both CPU and GPU via CUDA, ensuring that operations can be performed in both environments without requiring modifications to the codebase.

NFR-5 The software must be able to generate a 28×28 single channel sample under 1 second in a Google Colab environment (NVIDIA Tesla T4 GPU).

3.1.3 Use cases

In this section, we present the main use cases of the system. Figure 3.1 shows a complete diagram that captures all possible interactions between the user and the system. Below, we provide detailed descriptions of two typical workflows, labeled UC-1 and UC-2, which represent common execution flows for training models and generating samples, either unconditionally or with classifier-guided conditioning.

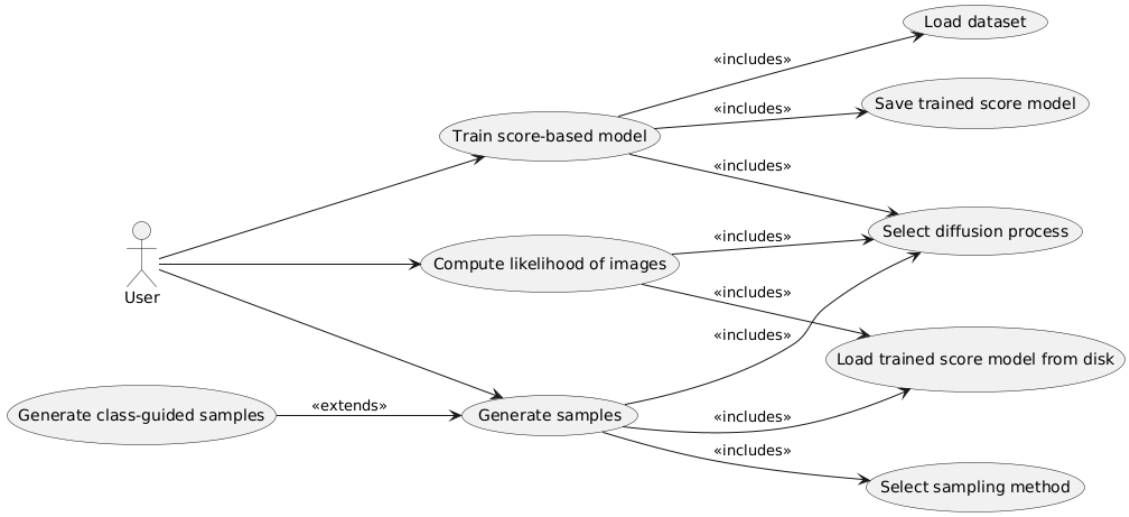


Figure 3.1: Use case diagram illustrating the main interactions of the user with the system

UC-1: Unconditional Sampling

This use case describes the complete process of training a score-based generative model on a given dataset and generating new images via unconditional sampling.

Actors: User of the software.

Preconditions: A dataset of unlabeled images is available in a supported format.

Main flow:

1. The user loads the dataset and specifies preprocessing parameters if needed.
2. The user selects an implemented diffusion process.
2. The system initializes and trains the score-based model on the dataset.
3. The trained model is saved to disk.
4. The user loads the trained score model from disk and specifies the desired sampling method.
5. The system generates initial noise vectors from the prior distribution and solves the reverse diffusion process specified by the user using the trained score model.
6. The resulting samples are returned and optionally saved or visualized.

Postconditions: A trained score model is available on disk and a batch of synthetic images has been generated.

UC-2: Classifier-Guided Sampling

This use case describes the process of training a score-based generative model and a classifier on a labeled dataset, and generating new images conditioned on a class label using classifier guidance.

Actors: User of the software.

Preconditions: A labeled dataset with consistent class annotations is available in a supported format.

Main flow:

1. The user loads the labeled dataset and specifies preprocessing parameters if needed.
2. The user selects an implemented diffusion process.
3. The system initializes and trains the score-based generative model on the dataset.
4. The trained score model is saved to disk.
5. The system trains a classifier to predict the class of noisy images from the same dataset.
6. The trained classifier is saved to disk.
7. The user loads both the trained score model and classifier from disk, specifies the target class label, and selects the desired sampling method.
8. The system generates initial noise vectors from the prior distribution and solves the guided reverse diffusion process using the classifier gradient and the trained score model.
9. The resulting class-conditional samples are returned and optionally saved or visualized.

Postconditions: A trained score model and classifier are available on disk, and a batch of class-conditional synthetic images has been generated.

3.2 Design

This section outlines the main design decisions made to implement the system's different functionalities. The overall architecture is organized into four main modules. The first is the diffusion module, which encapsulates all logic related to the definition and handling of diffusion processes, providing a flexible interface for implementing different variants. The second is the U-Net module, which contains the components of the neural network responsible for learning the score function. Finally, we describe the sampling module, which implements the various solvers used to generate images from the trained models, and the likelihood module, that allows for the computation of the log-likelihood of samples. A simplified class diagram containing the main modules and classes from the project is shown in Figure 3.2.

3.2.1 Diffusion processes

The `DiffusionProcess` is an abstract class responsible for modeling the diffusion process used in score-based generative models. Any implementation of a DiffusionProcess must have the following

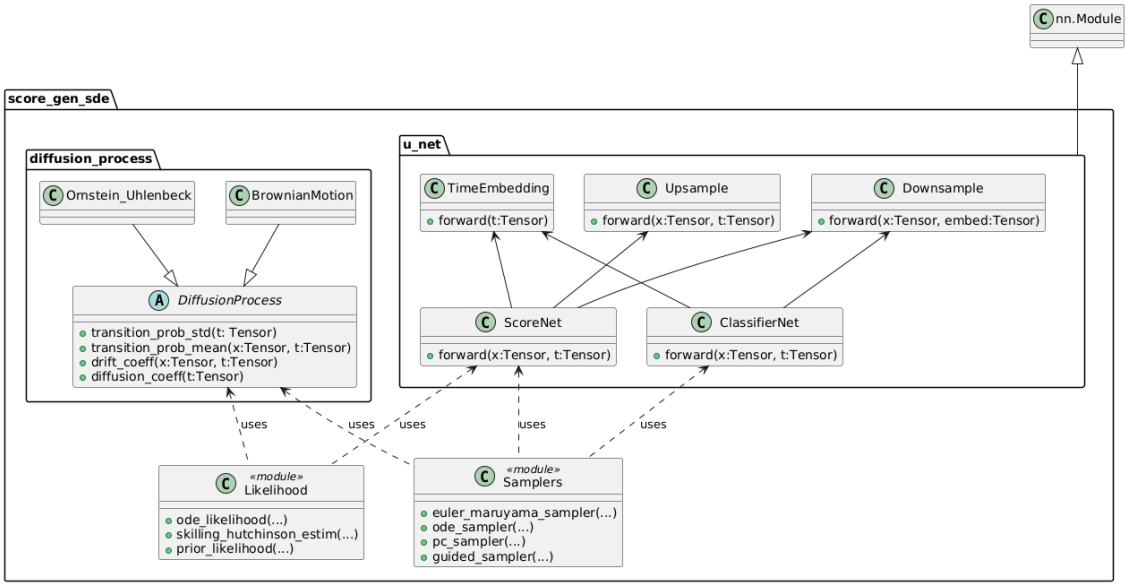


Figure 3.2: Simplified class diagram of the project. Arguments of some functions have been avoided for readability. All classes from `u_net` inherit from `nn.Module`

methods:

DP-1 `transition_prob_std(t: torch.Tensor)`: Computes the standard deviation of the transition probability $p_{0t}(\mathbf{x}(t) \mid \mathbf{x}(0))$ at a given time t.

DP-2 `transition_prob_mean(x: torch.Tensor, t: torch.Tensor)`: Computes the mean of the transition probability $p_{0t}(\mathbf{x}(t) \mid \mathbf{x}(0))$ at a given time t for input data x.

DP-3 `drift_coeff(x: torch.Tensor, t: torch.Tensor)`: Computes the drift coefficient of the diffusion process at a given time t for input data x.

DP-4 `diffusion_coeff(t: torch.Tensor)` Computes the diffusion coefficient of the diffusion process at a given time t.

Functions **DP-1** and **DP-2** are useful to add noise to an image, sampling from a standard Gaussian distribution and applying (2.8). On the other hand, functions **DP-3** and **DP-4** correspond with $f(t)$ and $g(t)$ in (2.6) and can be used by the samplers to model the reverse SDE (2.10) in order to reverse the process for sampling. A version of the Brownian Motion (2.7) with a diffusion coefficient $g(t) = \sigma^t$ has been implemented in the class `BrownianMotion`. This is the diffusion process that will be used throughout the experimental phase. Nonetheless, a version of the process in (2.9) has been included as well in the `OrnsteinUhlenbeck` class for possible further experimentation.

3.2.2 The U-Net

One of the key elements of the software is the U-Net, implemented in the `ScoreNet` class, used to estimate the score of images. Figure 3.3 exemplifies how a `ScoreNet` network works and its main operations. It receives as an argument a tensor x representing a mini-batch of images and a tensor t representing a time for each image. The shape of x is (N, C, W, H), where N is the batch size, C is the number of channels, and W and H are the width and height of the input image, while t has shape $(N, 1)$, where N is the batch size. The network returns a tensor of the same shape as x.

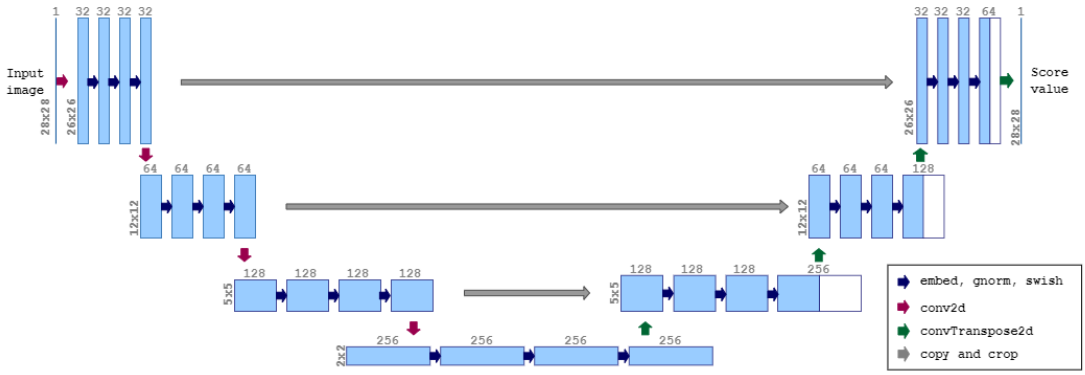


Figure 3.3: U-Net example for a 28x28 image with 1 channel. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

The main operations used by the `ScoreNet` are the following:

SN-1 `TimeEmbedding`: The embedding layer is a sequential layer that chains a Gaussian Fourier projection of the time tensor, a linear layer with the same size for the input and the output and finally a linear layer that maps the input to the desired number of channels. The Gaussian Fourier projection is an operation that projects the time tensor into a higher-dimensional space via Gaussian random features. Specifically, it samples $\omega \sim \mathcal{N}(\mathbf{0}, s^2\mathbf{I})$ which is subsequently fixed for the model (i.e., not learnable) and then, using t, it computes the corresponding Gaussian random feature defined as

$$[\sin(2\pi\omega t); \cos(2\pi\omega t)], \tag{3.1}$$

where $[\vec{a}; \vec{b}]$ denotes the concatenation of vector $\vec{a}$ and $\vec{b}$. This Gaussian random feature can be used as an encoding for time step t so that the score network can condition on t by incorporating this encoding.

SN-2 `GroupNorm`: The group normalization layer is a normalization layer that normalizes the input tensor along the channel dimension. It divides the input tensor into groups and

computes the mean and variance of each group separately. The input tensor is then normalized using the mean and variance of each group.

SN-3 `Swish`: The swish function is a non-linear activation function that is used to introduce non-linearity into the model. It is defined as $f(x) = x \cdot \sigma(x)$, where $\sigma(x)$ is the sigmoid function and therefore is defined as:

$$f(x) = \frac{x}{1 + e^{-x}} \tag{3.2}$$

The Swish function can be seen as a smooth version of the ReLU function. When approaching to absolutely larger numbers, the Swish function behaves like the ReLU function, mapping positives into themselves and negatives into zero

SN-4 `Conv2d`: The convolutional layer is the basic building block of the U-Net architecture. It is used to extract features from the input image. It uses a kernel to slide over the input image and compute the dot product between the kernel and the input image. In the `ScoreNet` implementation, convolutional layers with a kernel size of 3×3 and a stride of 1 and 2 are used. The number of channels in the output feature map is equal to the number of filters in the convolutional layer.

SN-5 `ConvTranspose2d`: The transposed convolution layer is used to upsample the input feature map. It uses a kernel to slide over the input feature map and compute the dot product between the kernel and the input feature map. The number of channels in the output feature map is equal to the number of filters in the transpose convolution layer.

The sequence of operations **SN-1**, **SN-2**, **SN-3**, and **SN-4** constitutes a downsampling step, which reduces the spatial dimensions of the image while increasing the number of feature channels. In contrast, the sequence **SN-1**, **SN-2**, **SN-3**, and **SN-5** performs the reverse operation and is referred to as an upsampling step. The `ScoreNet` architecture employs three downsampling and three upsampling steps. These sequences have been modularized into the `Downsample` and `Upsample` classes, respectively.

Although these are the fundamental operations of the network, there are additional architectural components that can significantly enhance model performance and expressiveness. Two particularly important additions are the following:

SN-6 `ResidualBlock`: Residual connections [25] are a key architectural feature that allow for better gradient flow in deep networks and facilitate the learning of identity mappings. In this implementation, a residual block consists of two sequences of normalization, activation, and convolution layers, with a learned time embedding added after the first convolution. If the input and output channels differ, a 1×1 convolution is applied to the input to match the dimensions. The final output is the sum of the input and the transformed path, scaled by a factor of $1/\sqrt{2}$ to preserve variance. This design improves the stability and convergence of

the network, especially in deep hierarchies.

SN-7 `SelfAttention2d`: To improrve the ability to capture long-range dependencies across spatial dimensions, a self-attention mechanism is employed [26]. The input is first normalized and reshaped into a sequence of tokens. Then values known as query, key, and value projections are computed via a 1×1 convolution, and attention weights are calculated through scaled dot-product attention. The output is then projected back and reshaped to its original spatial format. This operation allows the network to dynamically reweight features across the image and improves generative fidelity in high-variance regions.

These two modules are typically inserted within the downsampling and upsampling paths of the U-Net to enhance both local feature extraction and global contextual reasoning. They have been implemented independently of the other components, allowing for flexible configuration depending on the computational budget and target dataset complexity.

On the other hand, the `ClassifierNet` shares the same downsampling path as the `ScoreNet`, but instead of proceeding with upsampling, it concludes with an adaptive average pooling operation that reduces the spatial dimensions of each image to 1×1, effectively summarizing global information from each channel [27]. The resulting tensor is then flattened into a one-dimensional vector and passed through a fully connected layer that maps the extracted features to the corresponding number of output classes, each representing a possible label.

Training the network

The training of the score-based generative model is performed using a carefully designed optimization routine that ensures stability and convergence throughout training. The function `train_score_model` included in the training module wraps the whole training routine that will be used for experimenting.

The optimizer used is Adam [28]. The learning rate follows a cosine annealing schedule with a linear warm-up phase. This scheduling strategy allows for robust early training dynamics and a gradual learning rate decay that encourages convergence at later stages.To mitigate the risk of exploding gradients, gradient norms are clipped to a maximum value of 1.0 before each optimizer step.

Throughout training, average loss values are computed for each epoch. Additionally, estimates of the data log-likelihood are computed periodically using a fixed number of samples. Both loss and log-likelihood values are recorded in a log file, and model checkpoints are saved after every epoch to facilitate reproducibility and model evaluation.

3.2.3 Sampling and computation of log-likelihood

The software architecture must support all sampling approaches described in the previous chapter, including both SDE and ODE-based solvers for reversing the stochastic differential equation, including the option for classifier-guided sampling. Furthermore, the implementation must provide functionality for exact likelihood computation via the probability flow ODE, offering a unified and extensible framework for generation and evaluation.

Although all samplers share the same objective, their internal logic differs significantly. For this reason, each of them has been included as a separate function within the `sampler` module. All samplers must receive the following params as arguments:

- `score_model`: The score-based model.
- `diffusion_process`: Object representing the diffusion process, which provides the SDE to be reversed for sample generation.
- `num_channels`: Number of output channels per generated sample.
- `height`: Height in pixels of the generated samples.
- `width`: Width in pixels of the generated samples.
- `num_samples`: The number of samples to generate.
- `device`: The device to use for sampling (CUDA or GPU)

The different sampling approaches included in the system are the following:

S-1 `euler_maruyama_sampler`: Generates samples from the score-based model using the Euler-Maruyama method (2.12). It receives the following additional arguments:

- `num_steps`: Number of steps used by the algorithm. The step size Δt will be $1/$`num_steps`.

S-2 `ode_sampler`: Generates samples from the score-based model solving equation (2.13) using an ODE solver. The following are optional additional arguments; they correspond to the arguments with the same name that will be passed to the function `solve_ivp` from `scipy.integrate` [29]:

- `rtol`: Float specifying the relative tolerance. It controls the relative accuracy (number of correct digits of the solution).
- `atol`: Absolute tolerance. Controls the absolute accuracy (number of correct decimal places).
- `method`: String or `OdeSolver` object representing the integration method to use. Defauk ins 'RK45' which corresponds to Explicit Runge-Kutta method of order 5.

S-3 `pc_sampler`: Behaves exactly as **S-1** but in every step it uses Langevin dynamics as in equation (2.4) to obtain a better approximation of the solution.

S-4 `guided_sampler`: Algorithm proposed by [17] to exploit classifiers for sampling. It must receive:

- `classifier`: `ClassifierNet` object trained on noisy images.
- `grad_scale`: Parameter that controls the influence of the gradient of the classifier.

Moreover, samplers **S-1**, **S-2**, and **S-3** can optionally be used with classifier guidance using the conditioned score as in (2.15). This can be done by passing a classifier and optionally the gradient scale as arguments.

In addition, the `ode_likelihood` function, similarly to **S-3**, solves the equation in (2.17) to compute the likelihood in bits per dimension. It requires as input the tensor x_0, representing the data point for which the likelihood is to be computed, as well as the color depth, which denotes the number of bits used to encode each pixel. By default, this value is set to 8, consistent with the definition in (2.20).

3.3 Development methodology

This section describes the methodology followed during the software development process. It covers the overall organization of the project, the strategies adopted to ensure code quality, and the tools and practices used for documentation and testing. Together, these aspects contribute to the maintainability, reliability, and clarity of the implemented system.

3.3.1 Project organization

In this section, we describe the research and software development process followed throughout the duration of the project.

The project has been structured into two clearly defined phases preceded by an initial period that was devoted to building foundational knowledge in neural networks and deep learning, along with gaining proficiency in core Python libraries, crucial for the development of the entire software system. Weekly meetings were held from the beginning of the project to ensure steady progress and to address questions related to both theory and implementation. The Gantt diagram in Figure 3.4 shows the project timeline and the different tasks completed.

The first half was dedicated primarily to research, given the strong theoretical foundation required by score-based diffusion models. As these models represent a relatively recent advancement in generative modeling, a thorough review of the relevant literature was essential. Many of the challenges encountered during this stage were related to the mathematical framework rather than implementation details. Once

a solid theoretical understanding had been achieved, the second phase of the project began: the design and development of the software architecture, which put into practice the concepts studied during the research stage.

For version control and backup, the project has relied on the platform *GitHub* [30], which is built on top of the distributed version control system *git* [31]. It has been used to manage the source code throughout the development process, enabling proper tracking of changes, and safe storage of all code versions. GitHub also facilitated structured commits, branching strategies for experimentation, and rollback capabilities in case of errors, contributing to both reproducibility and codebase integrity.

3.3.2 Coding environment and libraries

This project has been developed using the Python programming language along with a set of specialized scientific computing and machine learning libraries. These tools were essential in implementing, training, and evaluating the score-based generative models described in this work.

NumPy [32] was used for general-purpose numerical computations, particularly for handling tensors and performing algebraic operations required in data preprocessing and model evaluation.

PyTorch [33] served as the core deep learning framework. The *torch.nn* module was used to define the components of the *ScoreNet* architecture, including convolutional layers, normalization, and activations. Training procedures were implemented using *torch.optim*, which provides various optimization algorithms, and *torch.utils*, which includes tools for data loading, batching, and parallel processing.

For numerical integration in the sampling and log-likelihood estimation routines, the project made use of *SciPy* [29], specifically its integration submodule.

To facilitate experiments on benchmark datasets, the *Torchvision* [34] package was employed, providing easy access to datasets such as MNIST and CIFAR-10, along with standard preprocessing transformations.

For model evaluation, *TorchMetrics* [35] has been used. In particular, to compute Fréchet Inception Distance (FID) and Inception Score (IS).

3.3.3 Code quality assurance

To ensure maintainability, readability, and coding standard compliance, we used a series of tools and practices that promote consistency, prevent errors, and support future development.

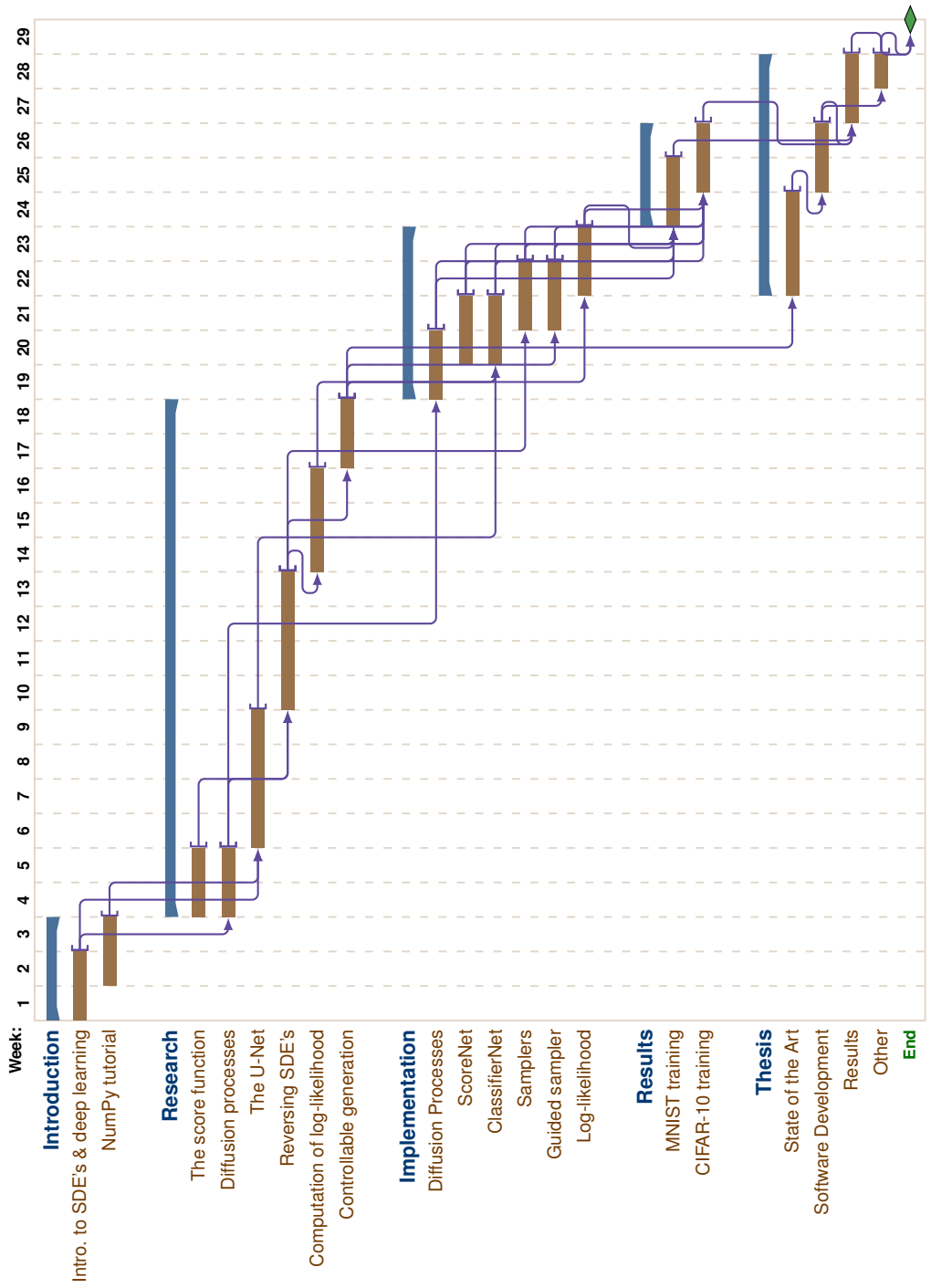


Figure 3.4: Gantt Diagram of The project. Weeks 1 to 17 correspond to the period between September 9th and December 20th, 2024. Week 18 begins on February 3rd, 2025.

Linting

To guarantee that the written code meets a certain style and adheres to standards, the following have been used:

- *Google Python Style Guide* [24]: The Google Python Style Guide provides a comprehensive set of conventions and best practices for writing Python code. It extends the standard *PEP 8* [36] guidelines by offering additional recommendations aimed at improving code clarity, consistency, and maintainability. Throughout this project, it has served as the primary reference to ensure stylistic uniformity across all modules.
- *pylint* [37]: A static code analysis tool that checks for possible errors and enforces coding standards. All modules in the project have been validated with it.

Formatting

Automatic formatters have been used to enforce consistent code formatting without manual intervention. The following tools have been integrated into the development workflow:

- *autopep8* [38]: Tool that automatically formats Python code to comply with *PEP 8* style guide. It has been employed to resolve specific warnings or for lightweight corrections of issues explicitly covered by *PEP 8*.
- *black* [39]: A more restrictive code formatter that enforces a uniform code style by applying a fixed set of rules. It was used as the default formatter throughout the project to standardize indentation, spacing, and line wrapping.

Type checking

All functions and class methods have been annotated with Python type hints to improve code clarity and reduce the likelihood of type-related errors. To ensure type safety throughout the codebase, the following tool has been used:

- *mypy* [40]: A static type checker for Python that verifies the consistency of type annotations at compile time. It has been run regularly to detect mismatches between expected and actual types.

3.3.4 Documentation

Clear documentation is key to making the software understandable, maintainable, and usable. In this project, we follow the official Google-style Python docstring conventions, with emphasis on inline comments and structured docstrings. Each function, method, and class includes a docstring describing

its purpose, inputs, outputs, and possible exceptions. This improves code readability and supports tools like Sphinx for automatic documentation generation [41].

An example of the documentation style used in the project is shown in Code 3.1. As illustrated, the docstring includes a summary line, followed by detailed descriptions of the parameters and return values.

Code 3.1: Example of docstring of function in the codebase.

```
def skilling_hutchinson_estim(
    f: Callable[[torch.Tensor, torch.Tensor], torch.Tensor],
    sample: torch.Tensor,
    time_steps: torch.Tensor,
    epsilon: torch.Tensor,
) -> torch.Tensor:
    """
    Compute the Skilling-Hutchinson estimator of the divergence:

    ϵ^⊤ J_f(x(t), t)ϵ

    Args:
        f: The function to estimate the Divergence of.
        sample: The input sample tensor.
        time_steps: The time steps tensor.
        epsilon: The epsilon tensor.

    Returns:
        The computed estimator.
    """
```

4

Results

4.1 Experimental environment

All experiments, including model training and sample generation, were conducted using `Google Colab Pro+` [42], a cloud-based computational environment that provides access to GPUs and a pre-configured Python environment suitable for deep learning. All reported runtimes and performance measurements in this section were obtained under this setup. The typical specifications of the Google Colab Pro+ environment are summarized in Table 4.1.

Resource	Specification
CPU	Intel(R) Xeon(R) CPU @ 2.20GHz
GPU	Nvidia Tesla K80, T4, L4, or P100 (depending on availability)
RAM	12.7 GB
Disk	235.7 GB

Table 4.1: Hardware and runtime specifications of the Google Colab Pro+ environment used for all experiments.

For simpler experiments, the NVIDIA Tesla T4 GPU was selected due to its high availability and lower costs. With a consumption rate of approximately 1.44 compute units per hour, it allows for up to 300 hours of GPU usage per month under the Colab Pro+ quota system. On the other hand, for more demanding tasks a NVIDIA Tesla L4 has been used, with a slightly higher consumption rate of 2.04 units per hour.

As Google Colab operates in a cloud-based environment, execution times are not necessarily reliable due to potential fluctuations in resource allocation. Nevertheless, in our experience with the Pro version, the runtime performance has proven to be highly consistent, with minimal variability observed across repeated executions.

4.2 MNIST dataset

The MNIST [43] (Modified National Institute of Standards and Technology) dataset consists of 70,000 grayscale images of handwritten digits (0–9), each of size 28×28 pixels, split into 60,000 training and 10,000 test samples. Due to its simplicity, MNIST has become a standard benchmark in computer vision, particularly for evaluating machine learning models. Although training advanced generative models is computationally demanding, MNIST allows an in-depth analysis without requiring extensive resources.

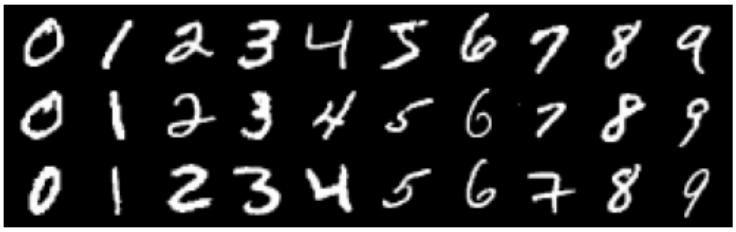


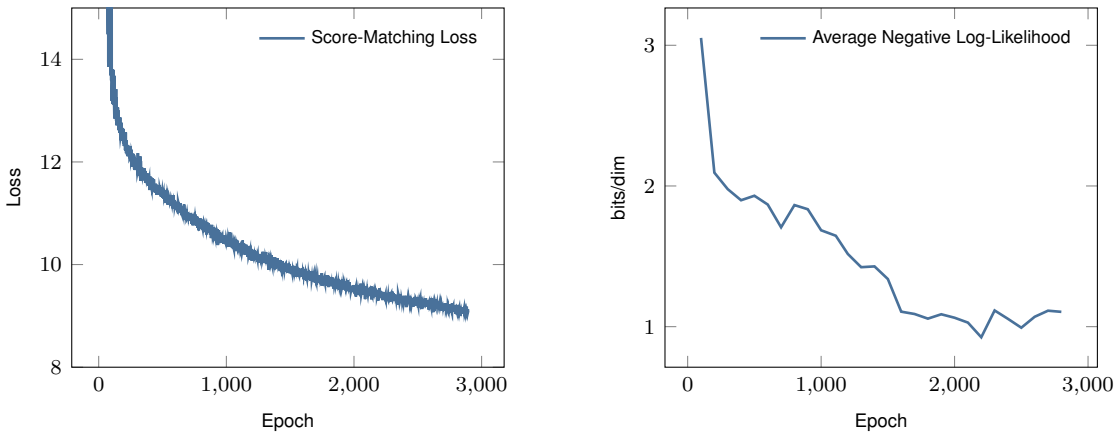
Figure 4.1: Batch of randomly picked samples from MNIST

A U-Net architecture has been implemented with four downsampling and four upsampling steps. Each convolutional layer is followed by a `ResidualBlock`, and a `SelfAttention2d` module is included at every intermediate level. The number of channels increases to 64, 128, and 256 in the first three downsampling steps, with the last step preserving dimensionality. The resulting model has approximately 7 million trainable parameters.

After exploring several hyperparameter combinations, the model was trained using a batch size of 128 and a learning rate of $2 \cdot 10^{-4}$. Figure 4.2(a) shows the evolution of the loss function over nearly $3{,}000$ training epochs. As observed in the plot, the loss decreases steadily, and convergence is not fully reached, indicating that further training could potentially improve performance. The training procedure lasted approximately 48 hours under the experimental conditions described above. Additional training was not conducted, as the results were good enough for the purposes of this work. All experiments in this section have been carried out using a NVIDIA Tesla T4.

Since loss minimization does not provide a direct intuition of how well the model captures the data distribution, the average negative log-likelihood (in bits per dimension) was also monitored. This was computed every 100 epochs on a batch of 1024 samples to reduce computational cost. While the metric is noisy and sensitive to outliers, it provides a useful heuristic. The evolution of the estimated log-likelihood is shown in Figure 4.2(b).

Once the score model has been trained, the negative log-likelihood (NLL) can be computed more rigorously across the entire dataset. Our model achieves an average NLL of 1.0848 bits per dimension, which is remarkably close to the theoretical lower bound. The mean entropy of the MNIST dataset has been estimated to be approximately 1.072 bits per dimension [44], reflecting the near-binary nature of



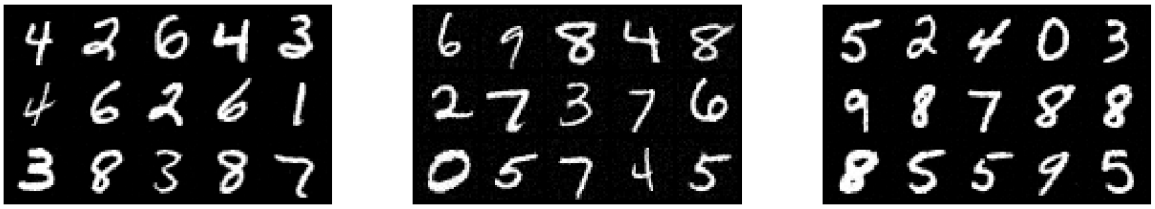
(a) Score-matching loss over training epochs.

(b) Estimated average log-likelihood in bits/dim per epoch.

Figure 4.2: Evolution of the training loss and average log-likelihood (in bits/dim) over epochs during model training.

the images. This result suggests that the model captures the data distribution with high fidelity. The negative log-likelihood was computed on a batch of 5,000 images, with integration carried out using the `RK45` method and parameters `rtol=1e-5` and `atol=1e-5`. All integration-related operations from this point forward shall be performed using these parameters.

Having obtained a reliable estimate of the score function, we generate samples using the methods described previously. Figure 4.3 displays a selection of images sampled from the trained model.



(a) Euler-Maruyama samples

(b) Probability flow ODE samples

(c) Predictor-Corrector samples

Figure 4.3: Samples generated using the score-based model trained on the MNIST dataset, with no cherry-picking. The samples in 4.3(a) were obtained using the Euler-Maruyama SDE solver with 1,000 sampling steps and the samples in 4.3(c) were produced using a Predictor-Corrector method, consisting of 1,000 Euler-Maruyama predictor steps and 1,000 Langevin corrector steps.

Although all samplers produce visually convincing samples, a quantitative evaluation is required. To this end, 5,000 samples were generated per sampler and evaluated using Fréchet Inception Distance (FID) and Inception Score (IS). Both metrics rely on features extracted by a pre-trained Inception network designed for high-resolution RGB images. Since MNIST images are grayscale and 28×28 pixels, they must be resized and duplicated across channels to match the expected input. Consequently, the computed scores reflect transformed rather than original samples, limiting their interpretability and

making them approximate indicators rather than precise comparisons. Table 4.2 reports the average sampling time per image, as well as the FID and IS scores for each sampler, under the experimental conditions described above.

Sampler	Time per sample (s)	FID	IS
Probability Flow ODE	0.07	82.63	2.09 ± 0.04
Euler-Maruyama	0.31	27.71	2.07 ± 0.04
Predictor-Corrector	0.31	45.82	2.04 ± 0.05

Table 4.2: Comparison of the three implemented samplers in terms of average sampling time, FID, and Inception Score. All metrics have been computed in a batch of 5000 images.

Among the evaluated methods, the Probability Flow ODE sampler is the fastest (0.07 s/sample) but yields the worst FID (82.63). The Euler-Maruyama sampler achieves the best FID (27.71), indicating higher sample fidelity at the cost of slower generation (0.31 s/sample). The Predictor-Corrector method shares the same runtime as Euler-Maruyama but results in a higher FID (45.82), suggesting a possible overcorrection or bias towards high-density regions of the data distribution, reducing sample diversity.

Inception Scores are similar across samplers, ranging from 2.04 to 2.09, with Probability Flow ODE slightly outperforming the others. As the score for real MNIST data is around 2.11 ± 0.05, all methods produce samples with comparable semantic quality and diversity.

The previous results correspond to unconditional sampling. To enable class-conditional generation, a classifier was trained on noisy images to guide the sampling process toward a target digit. Training was performed for 900 epochs (8 hours total) using a batch size of 128 and a learning rate of $2 \cdot 10^{-5}$ under previously described conditions. Once trained, the classifier allows class-conditional sampling with any of the four samplers, so the digit of the sample can be chosen. Figure 4.4 shows a batch of samples generated with classifier guidance using the guided sampler. Additional class-conditional samples obtained with the remaining sampling methods are presented in Figure A.1.

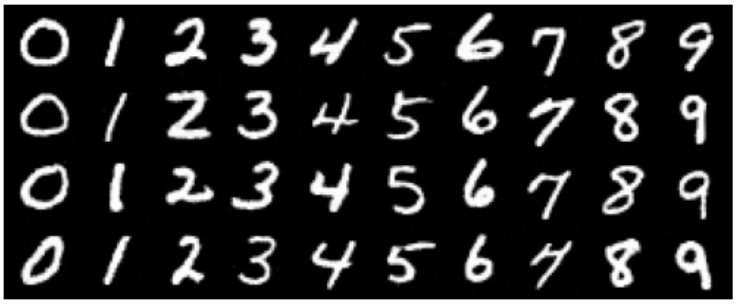


Figure 4.4: Samples generated using the score-based model trained on MNIST using the guided sampler

Samples generated using classifier guidance appear to be of even higher quality. To evaluate their performance against unconditional samples, we compute the same set of metrics, this time using

classifier-guided generation. The results are summarized in Table 4.3.

Sampler	Time per sample (s)	FID	IS
Euler-Maruyama	0.64	67.57	1.57 ± 0.02
Predictor-Corrector	0.64	92.54	1.49 ± 0.02
Probability Flow ODE	0.19	120.3	1.66 ± 0.02
Guided Sampler	0.64	41.5	2.09 ± 0.05

Table 4.3: Comparison of the three implemented samplers, using classifier guidance, in terms of average sampling time, FID, and Inception Score. All metrics have been computed in a batch of 5000 images.

The first observation is that sampling time increases significantly when classifier guidance is enabled, requiring more than twice the time per sample compared to the unguided setting. This overhead is primarily due to the additional gradient computations involved during sampling, which are often computationally expensive.

In terms of quality metrics, the standard samplers exhibit a general degradation in performance under classifier guidance: FID values increase substantially while Inception Scores decrease, indicating reduced distributional fidelity. This behavior is consistent with the fact that sampling from the theoretical conditional distribution $p(\mathbf{x} \mid y)$ encourages the generation of samples that are highly representative of class y, potentially at the expense of diversity. The notable exception is the guided sampler, which clearly outperforms the others by achieving a balanced trade-off, producing a relatively low FID of 41.5 and an Inception Score of 2.09. This suggests that the guided sampler not only benefits from classifier information but does so in a way that preserves both diversity and recognizability of the generated samples.

4.3 CIFAR-10 dataset

While MNIST offers an ideal setting for a first exploration of score-based generative modeling, its limited visual complexity motivates the use of a more challenging dataset. To this end, we extend our experiments to CIFAR-10 [45], a widely used benchmark in computer vision. CIFAR-10 consists of 60,000 color images of size 32×32 pixels, evenly distributed across 10 object categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Although CIFAR-10 images are still relatively low in resolution, they are considerably more complex than MNIST, containing three color channels and encoding more information. Reference models have captured its distribution at approximately 3 bits per dimension [46], nearly three times higher than for MNIST. This increased complexity demands more model capacity and longer training times. Given the limited computational resources available in this work, a full exploration is infeasible. Nonetheless, we aim to demonstrate that the model can capture complex data distributions effectively.

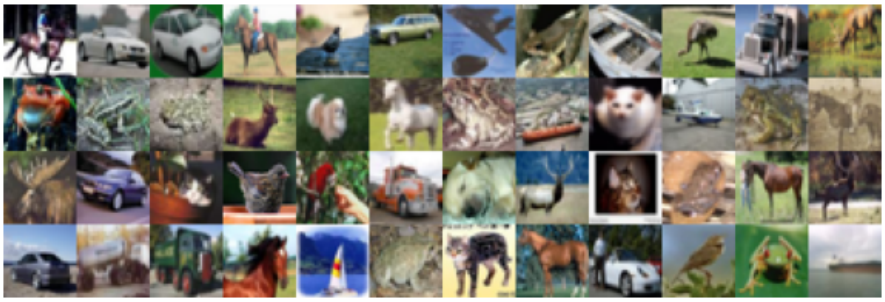
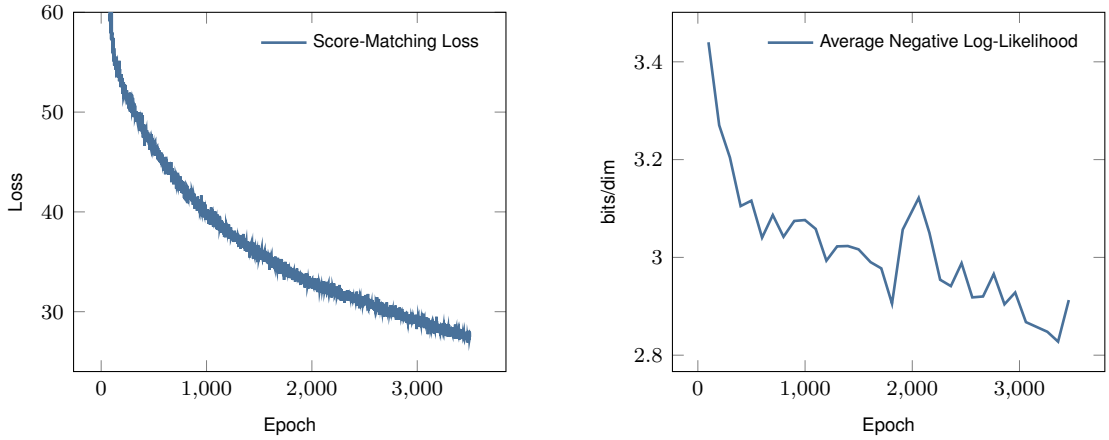


Figure 4.5: Batch of samples from CIFAR-10.

A more powerful U-Net architecture was designed for training on CIFAR-10, consisting of four resolution levels with a total of 32 residual blocks (16 downsampling, 16 upsampling) and a channel progression of $[128, 256, 256, 256]$. Self-attention layers were added at intermediate resolutions to capture long-range dependencies. The model was trained for over 3,500 epochs with a batch size of 128 and a learning rate of $2 \cdot 10^{-4}$, during a total time of 48 hours. All training and sampling operations in this section have been carried out using an NVIDIA Tesla L4 GPU. Figure 4.6 shows the evolution of the loss and negative log-likelihood.



(a) Score-matching loss over training epochs.

(b) Estimated average log-likelihood in bits/dim per epoch.

Figure 4.6: Evolution of the training loss and average log-likelihood (in bits/dim) over epochs during model training.

To evaluate how well the model captures the data distribution, the log-likelihood was computed on a subset of $1{,}024$ test samples, yielding an average of 2.96, which is close to the expected lower bound. However, a higher value of bits per dimension does not necessarily imply good perceptual quality.

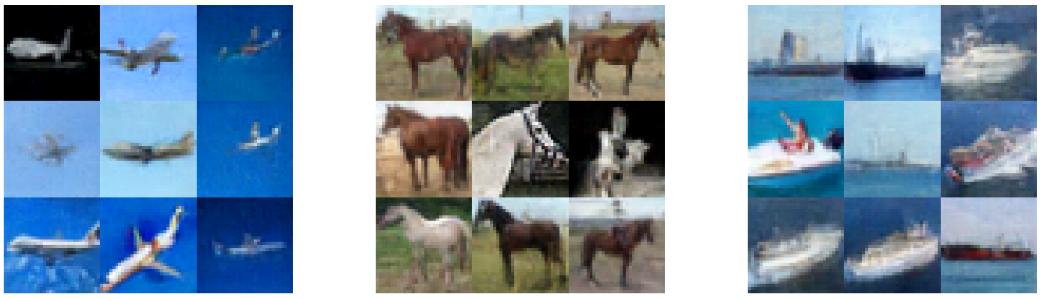
Figure 4.7 displays generated samples. While some features of the original dataset are preserved, the model struggles to clearly distinguish between classes, likely due to the greater separation between class distributions in CIFAR-10 compared to MNIST. To address this, classifier guidance is introduced. As in previous sections, a classifier is trained on noisy images to steer the generation toward a desired

class.

(a) Euler-Maruyama samples (b) Probability flow ODE samples (c) Predictor-Corrector samples

Figure 4.7: Samples generated using the score-based model trained on the CIFAR-10 dataset, with no cherry-picking. The samples in 4.7(a) were obtained using the Euler-Maruyama SDE solver with 1,000 sampling steps. Samples in 4.7(c) were produced using a Predictor-Corrector method, consisting of 1,000 Euler-Maruyama predictor steps and 1,000 Langevin corrector steps.

We now incorporate the classifier into the sampling process to generate guided samples. Selected results are shown in Figure 4.8, with additional samples from other classes also available in Figure A.2. The images suggest that classifier guidance improves visual quality and class distinguishability, although potentially at the cost of reduced sample diversity. Since the model struggles to fully capture the class variability in CIFAR-10, retraining it on a single class may help determine if the limitations are due to class variability and whether a narrower focus improves generation.

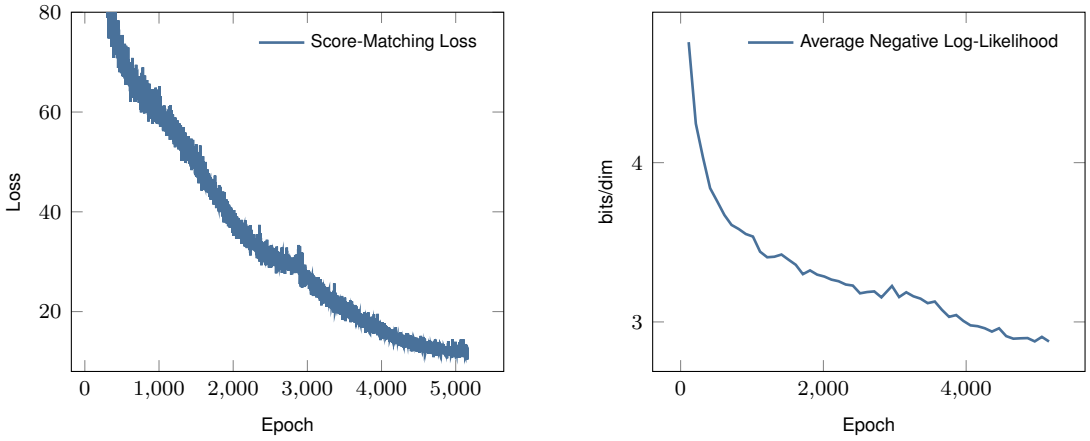


(a) Plane (b) Horse (c) Ship

Figure 4.8: Samples generated using the score-based model trained on the CIFAR-10 dataset, with no cherry-picking. All samplers have been drawn using the `guided_sampler` method with a `grad_scale` level of 5.

To this end, we follow the same approach, this time restricting the training data to the automobile class only. Given the reduced number of samples, the training process is significantly faster, allowing for a more extended training schedule. Figure 4.9 illustrates the training evolution over approximately $5,000$ epochs, corresponding to a total training time of 8 hours.

A potential issue when working with smaller datasets is the increased risk of overfitting. To verify that the model does not suffer from this problem, we evaluate its negative log-likelihood on the test set



(a) Score-matching loss over training epochs.

(b) Estimated average log-likelihood in bits/dim per epoch.

Figure 4.9: Evolution of the training loss and average log-likelihood (in bits/dim) over epochs during model training.

and compare it to the one computed on the training set. Specifically, we train the model on car images and compute the likelihood over unseen test samples. As shown in Figure 4.9(b), the average NLL on the training set is approximately 3 bits per dimension, while the corresponding value on the test set is slightly higher at 3.61. Although this difference suggests a mild degree of overfitting, the likelihood remains within a reasonable range, indicating that the model still generalizes adequately to unseen data.

Samples generated by the model using the predictor-corrector sampler are presented in Figure 4.10. Samples from the other samplers can be found in A.3. As expected, both the visual quality and the diversity of the samples improve noticeably. To evaluate this quantitatively, we replicate the experiments from the previous section and compute the Inception Score (IS) and Fréchet Inception Distance (FID) on the generated data. In this case, we evaluate 5,000 samples from the model against the 5,000 training samples from the dataset. The results are summarized in Table 4.4.

Figure 4.10: Batch of generated samples from CIFAR-10 automobile class using a predictor-corrector sampler with 1000 predictor steps and 1000 corrector steps.

Again, the Probability Flow ODE method exhibits the lowest sampling time, generating samples at an

Sampler	Time per sample (s)	FID	IS
Probability Flow ODE	0.14	164.29	4.41 ± 0.10
Euler-Maruyama	0.34	50.75	3.57 ± 0.08
Predictor-Corrector	0.34	49.07	3.32 ± 0.14

Table 4.4: Comparison of the three implemented samplers in terms of average sampling time, FID, and Inception Score. All metrics have been computed in a batch of 5000 images.

average of 0.14 seconds per image. However, this efficiency comes at the cost of significantly reduced fidelity, as reflected by its FID score of 164.29, the highest among the three methods. In contrast, both the Euler-Maruyama and Predictor-Corrector samplers achieve substantially lower FID values (50.75 and 49.07, respectively), indicating better alignment with the training distribution. Nevertheless, this improvement requires over twice the sampling time.

On the contrary, the Probability Flow ODE method achieves the highest IS (4.41), its corresponding FID suggests that these samples may be confidently classifiable yet less diverse. Clearly, the methods based on SDE resolution provide a more favorable balance between sample quality and diversity.

5

Conclusions and future work

This project has explored the theoretical foundations and practical implementation of score-based generative models formulated through stochastic differential equations (SDEs). We developed a modular software framework that enables the training and sampling of images through diffusion models using methods for score estimation. The implementation covers the full generative pipeline, including the definition of the forward process, the design of a time-dependent U-Net architecture for score estimation, the training routine, and a family of sampling methods tested on benchmark datasets such as MNIST and CIFAR-10.

The experimental results display the behavior of the proposed framework, showing its ability to generate coherent samples. Although the visual fidelity of the outputs is naturally constrained by the simplicity of the architecture and limited training resources, the project is designed to enable reproducibility and facilitate future experimentation.

Although this project and much of the recent literature have focused on image generation, the potential applications of score-based generative models extend well beyond this domain. For instance, in the field of molecular modeling, diffusion models have been used to sample valid 3D molecular conformations with symmetry constraints and physical priors [47, 48]. In audio generation, models such as *DiffWave* [49] have successfully synthesized high-quality waveforms from white noise signals using a score-based approach. In medical imaging, these models are being explored for data augmentation, denoising, and image reconstruction [50]. Score-based models have also shown promise in point cloud generation [51], protein structure modeling [52], and inverse problems such as MRI reconstruction [53].

On the other hand, the capabilities and limitations of score-based models can be further explored through the study of alternative diffusion processes. The current implementation relies on standard one-dimensional diffusion processes for noise perturbation. However, recent studies have introduced more advanced formulations, such as the *critically damped Langevin diffusion* [54], which have demonstrated faster convergence rates and improved sampling efficiency in high-dimensional generative tasks.

Another potential improvement could involve using more expressive neural architectures. While this project employs a relatively simple U-Net to estimate the score function, state-of-the-art results in the field are often achieved with deeper architectures that incorporate advanced components such as

transformer-style modules.

In addition, evaluating the model on more complex datasets would help assess its generalization capabilities. The current experiments are limited to simpler datasets due to computational constraints, but extending the analysis to higher-dimensional datasets, such as CelebA-HQ [55] or subsets of ImageNet [56], would provide a more realistic test of performance and reveal architectural or optimization limitations not evident at lower resolutions.

Finally, a further research direction involves incorporating uncertainty quantification techniques. Recent theoretical advances have shown that score-based models can provide meaningful confidence estimates by analyzing how approximation errors in the score function propagate through the generative process [57]. Using these tools can improve model interpretability and reliability, especially in fields where trust is crucial, like medical imaging or scientific simulation.

Bibliography

[1] A. Bora, A. Jalal, E. Price, and A. G. Dimakis, "Compressed sensing using generative models," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, vol. 70, pp. 537–546, 2017.

[2] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 2223–2232, 2017.

[3] Y. Song, L. Shen, L. Xing, and S. Ermon, "Solving inverse problems in medical imaging with score-based generative models," in *International Conference on Learning Representations (ICLR)*, 2022.

[4] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[5] D. J. Rezende and S. Mohamed, "Variational inference with normalizing flows," in *International Conference on Machine Learning*, pp. 1530–1538, PMLR, 2015.

[6] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, 2014.

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.

[8] Y. Song, "Generative modeling by estimating gradients of the data distribution." `https://yang-song.net/blog/2021/score/`, 2021. Accessed: 2025-02-27.

[9] Y. Song and S. Ermon, "Generative modeling by estimating gradients of the data distribution," *CoRR*, vol. abs/1907.05600, 2019.

[10] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, "Score-based generative modeling through stochastic differential equations," 2021.

[11] A. Hyvärinen, "Estimation of non-normalized statistical models by score matching," *J. Mach. Learn. Res.*, vol. 6, p. 695–709, Dec. 2005.

[12] P. Vincent, "A connection between score matching and denoising autoencoders," *Neural Computation*, vol. 23, no. 7, pp. 1661–1674, 2011.

[13] Y. Song and S. Ermon, "Improved techniques for training score-based generative models," *CoRR*, vol. abs/2006.09011, 2020.

[14] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[15] B. D. Anderson, "Reverse-time diffusion equation models," *Stochastic Processes and their Applications*, vol. 12, no. 3, pp. 313–326, 1982.

[16] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, vol. 23 of *Applications of Mathematics*. Springer, 1992.

[17] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," 2021.

[18] J. Skilling, *The Eigenvalues of Mega-dimensional Matrices*, pp. 455–466. Dordrecht: Springer Netherlands, 1989.

[19] M. Hutchinson, "A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines," *Communication in Statistics- Simulation and Computation*, vol. 18, pp. 1059–1076, 01 1989.

[20] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, vol. 29, pp. 2234–2242, 2016.

[21] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *Advances in neural information processing systems*, vol. 30, pp. 6626–6637, 2017.

[22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[23] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

[24] Google, "Google Python Style Guide." `https://google.github.io/styleguide/pyguide.html`, 2025. Accessed: 2025-03-25.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

[27] V. Sandhu, "Guided diffusion by openai from scratch." `https://www.kaggle.com/code/vikramsandu/guided-diffusion-by-openai-from-scratch`, 2023. Accessed: 2025-05-11.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[29] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[30] "GitHub." `https://github.com/`, 2024. Accessed: 2025-03-25.

[31] "Git." https://git-scm.com/, 2024. Accessed: 2025-03-25.

[32] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[34] S. Marcel and Y. Rodriguez, "Torchvision: Datasets, transforms and models specific to computer vision." https://pytorch.org/vision/stable/index.html, 2010. Accessed: 2025-05-03.

[35] Lightning-AI, "Torchmetrics: Machine learning metrics for pytorch." https://github.com/Lightning-AI/torchmetrics, 2022. Accessed: 2025-03-24.

[36] G. van Rossum, B. Warsaw, and A. Coghlan, "Pep 8 – style guide for python code." https://peps.python.org/pep-0008/, 2001. Accessed: 2025-03-25.

[37] PyCQA, "Pylint – python static code analysis." https://pypi.org/project/pylint/, 2025. Accessed: 2025-03-25.

[38] H. Hattori, "autopep8." https://pypi.org/project/autopep8/, 2025. Accessed: 2025-03-25.

[39] Łukasz Langa and contributors, "Black: The uncompromising python code formatter." https://black.readthedocs.io/en/stable/, 2025. Accessed: 2025-03-25.

[40] J. Lehtosalo and contributors, "Mypy: Optional static typing for python." https://mypy-lang.org/, 2025. Accessed: 2025-03-25.

[41] A. Turner, B. Tran, C. Sewell, F. Freitag, J. L. Andersen, J.-F. Burnol, S. Finucane, T. Shimizukawa, and T. Komiya, "Sphinx documentation." http://sphinx-doc.org/sphinx.pdf, 2024. Accessed: 2025-03-25.

[42] Google, "Google colaboratory." https://colab.research.google.com/, 2025. Accessed: 2025-05-14.

[43] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[44] M. Deng, S. Li, I. Kang, N. Fang, and G. Barbastathis, "On the interplay between physical and content priors in deep learning for computational imaging," 04 2020.

[45] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., University of Toronto, 2009. Technical report, CIFAR.

[46] Y. Song, C. Durkan, I. Murray, and S. Ermon, "Maximum likelihood training of score-based diffusion models," 2021.

[47] E. Hoogeboom, V. G. Satorras, C. Vignac, and M. Welling, "Equivariant diffusion for molecule generation in 3d," in *International Conference on Machine Learning*, pp. 8867–8887, PMLR, 2022.

[48] M. Xu, S. Luo, A. Bietti, and T. Jaakkola, "Geodiff: a geometric diffusion model for molecular

conformation generation," in *International Conference on Learning Representations*, 2022.

[49] Z. Kong, W. Ping, J. Huang, K. Zhao, and B. Catanzaro, "Diffwave: A versatile diffusion model for audio synthesis," *arXiv preprint arXiv:2009.09761*, 2020.

[50] A. Jalal, Y. Song, A. Hannun, R. Grosse, and S. Ermon, "Robust compressed sensing mri with learned diffusion models," in *Advances in Neural Information Processing Systems*, vol. 34, pp. 14938–14954, 2021.

[51] S. Luo, P. Hu, R. Liao, A. Torralba, S. Fidler, and L. Zhang, "Diffusion probabilistic models for 3d point cloud generation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2837–2846, 2021.

[52] J. L. Watson, D. Juergens, N. R. Bennett, B. L. Trippe, J. Yim, H. E. Eisenach, *et al.*, "Broadly applicable and accurate protein structure prediction with rosettafold," *Nature*, vol. 600, no. 7889, pp. 701–705, 2022.

[53] H. Chung, S.-H. Sim, J. Yoo, and J. C. Ye, "Score-based diffusion models for accelerated mri reconstruction," *arXiv preprint arXiv:2110.05243*, 2022.

[54] T. Dockhorn, A. Vahdat, and K. Kreis, "Score-based generative modeling with critically-damped langevin diffusion," 2022.

[55] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," in *International Conference on Learning Representations*, 2018.

[56] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, IEEE, 2009.

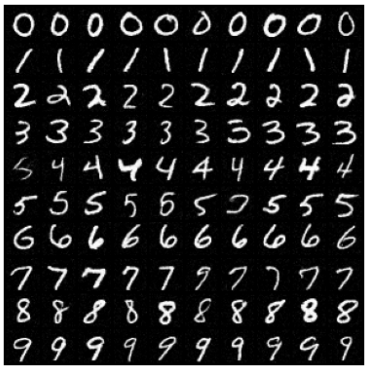
[57] G. Mimikos-Stamatopoulos, K. Reddy, A. G. Wilson, and R. B. Grosse, "Score-based generative models are provably robust: an uncertainty quantification perspective," *arXiv preprint arXiv:2405.15754*, 2024.

APPENDICES

A

Generated Samples

(a) Euler-Maruyama samples with guidance

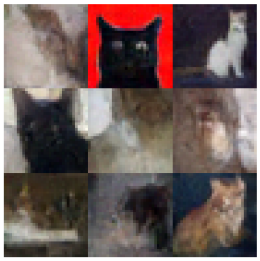


(b) Probability flow ODE samples with guidance

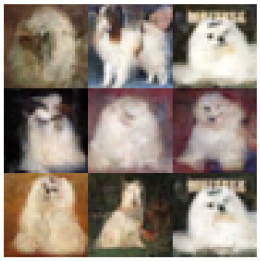
(c) Predictor-Corrector samples with guidance

(d) Classifier guided samples with guidance

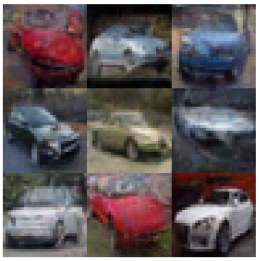
Figure A.1: Samples generated using the score-based model trained on the MNIST dataset, with no cherry-picking. The samples in A.1(a) were obtained using the Euler-Maruyama SDE solver with 1,000 sampling steps. Samples in A.1(c) were produced using a Predictor-Corrector method, consisting of 1,000 Euler-Maruyama predictor steps and 1,000 Langevin corrector steps. Lastly samples in A.1(d) were drawn using the generic classifier-guided sampler with 1,000 steps. A `grad_scale` of 1 was used for A.1(c) and A.1(d) while it was set to 1.5 for A.1(a) and 2 for A.1(b)



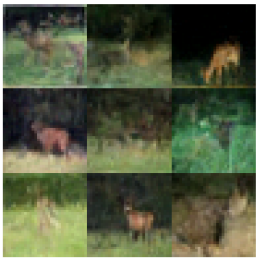
(a) Cat



(b) Dog



(c) Automobile



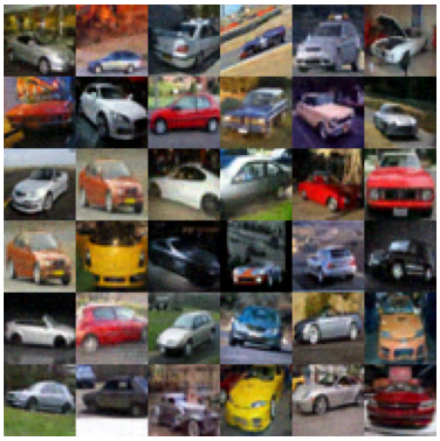
(d) Deer

(e) Frog

(f) Truck

Figure A.2: Samples generated using the score-based model trained on CIFAR-10, with no cherry-picking, using the `guided_sampler` method with a `grad_scale = 5`.

(a) Euler-Maruyama samples with guidance



(b) Probability flow ODE samples with guidance

Figure A.3: Samples generated using the score-based model trained on CIFAR automobile class, with no cherry-picking. Samples in A.3(a) were obtained using 1,000 sampling steps. Samples in A.3(b) were generated using parameters `rtol=1e-5`, `atol=1e-5`, and method `RK45`.


UÁM
Universidad Autónoma
de Madrid