Project 4: Generative Modeling using Diffusion Models

CSE 849 Deep Learning (Spring 2025)

Gautam Sreekumar
Instructor: Zijun Cui

April 13, 2025

In this project, we will use diffusion models for unconditional and conditional sample generation. Since naively implemented diffusion models require a lot of memory, we will work on a synthetic 2-dimensional dataset called "STATES", shown in Fig. 2. Although the models used in this project are lightweight compared to the diffusion models used in practice, they still require a considerable amount of GPU memory and time. So please start this assignment early and use HPCC if needed.

1 Introduction

Diffusion models (DMs) were introduced by Sohl-Dickstein et al. [Soh+15] to model arbitrary probability distributions by framing the task as analogous to a thermodynamic gas diffusion process. In this analogy: some gas particle are initially distributed along a 2D contour (e.g., the outline of a duck's body as shown in Fig. 1) within a vacuum cube. As time progresses, these particles gradually diffuse and occupy the entire space. However, the exact trajectory of each particle may be unpredictable since we do not know everything about this system. Our probabilistic modeling framework mirrors this gas system: the gas particles represent the *samples* we wish to generate, the 2D contour represents the target data distribution p_{data} that we wish to model; and the fully diffused gas follows a simple, known probability distribution p_{init}. The diffusion process where the gas particles occupy the vacuum is called the forward diffusion process, and pulling all the free-floating gas particles back to the contour is the reverse diffusion process. In probabilistic modeling, this forward diffusion process is equivalent to simply adding noise until p_{data} is indistinguishable from p_{init} which can be described analytically. A DM learns the reverse diffusion process that transforms p_{init} back to p_{data}.

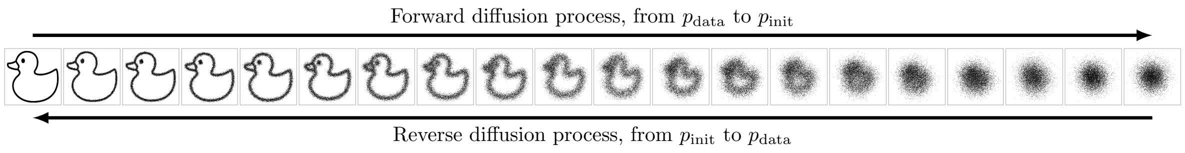
Forward diffusion process, from p_{data} to p_{init}
Reverse diffusion process, from p_{init} to p_{data}

Figure 1: Each black dot represents a gas particle. Initially, the particles are distributed along the duck's contour, corresponding to the data distribution p_{data}. Over time, they diffuse away from the contour and spread throughout the vacuum around it. Eventually, their distribution becomes indistinguishable from a simple Gaussian distribution p_{init}.

The real potential of DMs was revealed by later works [SE19; HJA20; Son+21] that posed the problem of finding p_{data} as a series of denoising steps that progressively removed the noise added during the forward diffusion process. During training, we will learn a denoising model that will take as inputs a noisy intermediate sample and the corresponding time step and output a cleaner sample for the next time step. During inference, we will start from the known distribution p_{init} and slowly denoise at every time step using our trained model to finally obtain a clean sample from p_{data}.

Additional Resources: Since DMs are highly popular right now, you can find several online resources that summarize the vast literature that covers various aspects of DMs. A few suggestions are lecture notes from the MIT course on diffusion models, Lilian Weng's blog, and Yang Song's blog.

If you are familiar with pre-DM deep probabilistic models like VAE, you can check this paper to know what exactly made DMs a great player in an already-crowded turf of deep generative models. Also, check Sec. 7.1 for a detailed discussion on what various terms in the diffusion model literature mean.

2 Task 1: Unconditional Generation

The first task in this project will be to implement a diffusion model (DM) for unconditional generation. Specifically, we will train a model to generate samples from STATES dataset. As you can see in Fig. 2, STATES consists of 2-dimensional data points forming the outlines of five US states - Ohio, Wisconsin, Oklahoma, Idaho, and Michigan (clockwise from top). We will denote the probability distribution function (PDF) of this dataset as $p_{\text{data}}(x, y)$, where x and y are the location of a sample and the label of the state to which that sample belongs, respectively. In the task of unconditional generation, your objective is to generate samples from $p_{\text{data}_t}(x)$. Here, the model is unaware of any labels y.

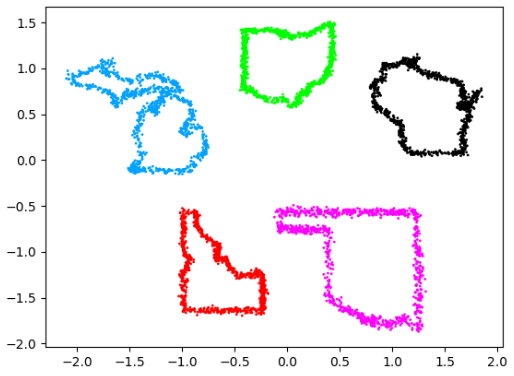

Figure 2: Example samples from "STATES" dataset in this project.

Step-by-step instructions In the following instructions, the notation follows lecture slides.

1. **Components**: Your denoising model will denoted by ϵ_θ where θ indicates that it is a parameterized. In contrast, the noise that this denoising model tries to estimate will be denoted simply by ϵ. The timestep will be denoted by t and will be measured along the forward diffusion process. That is, $t = 0$ corresponds to p_{data} and $t = T$ corresponds to p_{init}. In other words, x_0 comes from p_{data} and x_T comes from p_{init}.

2. **Forward process**: At every timestep t, you can estimate x_t from x_{t-1}:

$$x_t \sim \mathcal{N}(\sqrt{1-\beta_t} x_{t-1}, \beta_t I), \tag{1}$$

where β_t adjusts the amount of noise added at each stage. A higher amount of noise will not make much difference as $t \to T$. Therefore, it is okay to have $\beta_{t_{\text{large}}} > \beta_{t_{\text{small}}}$. In general, $0 < \beta_1 < \beta_2 < \cdots < \beta_T < 1$. From Eq. (1), we can derive the relationship between variances of the random variables involved in it.

$$\text{var}(x_t) = (1 - \beta_t)\text{var}(x_{t-1}) + \beta_t$$

If the variance of the real data is $\text{var}(x_0) = 1$, then $\text{var}(x_1) = \text{var}(x_2) = \cdots = \text{var}(x_T) = 1$, irrespective of the value of β_t. Thus, this is a variance-preserving transformation. Given a sample x_{t-1}, to obtain a sample x_t, we can simply add noise to a scaled x_{t-1}:

$$x_t = (1 - \beta_t)x_{t-1} + \beta_t\epsilon,$$

where $\epsilon \sim \mathcal{N}(0, I)$ (`torch.randn` function). This process is very inefficient since you need to repeat the same computation every timestep. Luckily, since the added noise values are i.i.d. at every timestep, you can approximate with a single noise value sampled from the standard normal distribution. Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$. Then, we can write,

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon. \tag{2}$$

In your dataloader, you will implement Eq. (2) to get x_t for arbitrary values of t.

3. **Backward process**: We will follow [HJA20] and train a denoising model. We will use an MLP as our denoising model. The input to this MLP will be the noisy sample $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$ and the corresponding timestep t, and it will predict $\hat{\epsilon}$. We will train this model using MSE loss: $\mathcal{L}_{\text{train}} = ||\epsilon - \hat{\epsilon}||_2^2$.

4. **Sampling**: To sample from p_{data}, we will start from a pure noise sample from $p_{\text{init}} = \mathcal{N}(0, I)$ and then progressively denoise it as following:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + z_t,$$

where z_t is sampled from $\mathcal{N}(0, I)$ at every timestep of sampling. After iterating through T timesteps, x_0 is your sample.

3 Task 2: Conditional Generation

In conditional generation, the task is to generate from the conditional distribution $p_{\text{data}}(x \mid y)$. As you may recall $p_{\text{data}}(x) = \sum_x p_{\text{data}}(y \mid x) p_{\text{data}}(x)$. $p_{\text{data}}(y \mid x)$ is a *classifier* as it gives us the probability for a given sample x to belong to label y. Note that $p_{\text{data}}(x)$ is the same PDF that we learned for unconditional generation. Thus, by learning an additional classifier $p_{\text{data}}(y \mid x)$, we will reuse our previous model to generate samples from each US state.

Step-by-step instructions For conditional generation, the training scheme is the same as that of the unconditional generation; only sampling is different.

1. **Training a classifier**: For sampling, we need a classifier $f_\phi(y \mid x_t)$ whose gradients can be calculated. Therefore, we will train an MLP that can successfully predict the US state label.

2. **Sampling**: At every timestep, compute the following:

$$\hat{\epsilon} = \epsilon_\theta(x_t) - \sqrt{1 - \hat{\alpha}_t} \nabla_{x_t} \log f_\phi(y \mid x_t)$$
$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1 - \bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1 - \bar{\alpha}_{t-1}}\hat{\epsilon}$$

4 Implementation Details

1. **Diffusion parameters**: Use $T = 500$ timesteps. We will use a linear β schedule with $\beta_0 = 10^{-4}$ and $\beta_T = 0.02$. Use `torch.linspace`.

2. **Training details**: Use a very large batch size (say, $> 10,000$). Suppose the dataset contains N samples from p_{data}. Then, the entire dataset contains NT samples, each created by adding noise corresponding to a specific timestep to a specific sample. Although NT is a large number, you can load the entire dataset to the GPU for faster training. The starter code is written for this setting to save running time. But since the noise added at every timestep is a random variable, you need to "refresh" the data after every, say, 100 epochs. The skeleton code for this is provided and will explain what to be done.

3. **Denoising model**: The denoising model is an MLP with 4 hidden layers, each with 256 hidden units. Its input is the noisy sample x_t and the timestep t concatenated to become a 3-dimensional input. Normalize the value of t as $\bar{t} = 2(t - T/2)/T$. The output is the 2-dimensional noise estimate.

4. **Classifier**: The classifier is an MLP with 3 hidden layers with 100, 200, and 500 hidden units, respectively. The output of this model is the 5-class logit and is passed on to the cross-entropy loss function for training. However, during conditional generation, you must pass the logits through a log-softmax layer (described in the next bullet-point). Since the classifier must provide gradients for the predicted labels given the noisy sample, you must also train the classifier with both the noisy sample as well as the corresponding timestep. Use the same $T = 500$ as the diffusion model. To evaluate your classifier prediction qualitatively, your classifier training code will generate a prediction map. Other details for training the classifier are not provided since you should be familiar with training an MLP by now.

5. **Conditional generation**: As described before, you require the gradients of the log-pdf from the classifier. Therefore, you must pass the classifier logits through a log-softmax layer before computing the gradients. Suppose your model's output after log-softmax is `out`. `out` is of shape $b \times 5$ where b is the batch size. Say you wish to generate samples with label 2. Then, take the column corresponding to this label and let's call it `out_label`. `out_label` is of shape b (the second dimension collapsed). To compute the gradients of the log-pdf from the classifier, apply the backward method on `out_label` with "gradient" argument set to b-shaped tensor containing ones (same shape as `out_label`). Do not use `torch.no_grad()` during this time. Since gradients of the classifiers are computed during sampling, it may require a large GPU memory. To avoid this, you can sample in whatever batchsize that your GPU can handle and concatenate them before computing negative log-likelihood (NLL).

5 Submission Deliverables and Formats

5.1 Deliverables

Submit all the required Python files and the report containing the following:

1. Scatter plot showing unconditional and conditional generation of samples. For conditional generation, include samples corresponding to each state. Generate 5000 samples for unconditional generation and 5000 samples for each state in conditional generation.

2. Training and validation curves generated by the programs for unconditional generation, and classifier training. Conditional generation does not involve training. You will reuse your model from the unconditional generation task.

3. Classifier prediction plot.

5.2 Formats

Submit a folder named `first name_last name_project_4` using the same subfolders as below:

1. code
   - `uncond_gen.py`
   - `cond_gen.py`
   - `data.py`
   - `model.py` – this file need not be modified and will not be graded.
   - `classifier.py`

2. outputs
   - `report.pdf`
   - `uncond_gen_samples.pt`: containing 5000 samples from unconditional generation.
   - `cond_gen_samples_0.pt`, ..., `cond_gen_samples_4.pt`: 5 files containing conditional generated samples from each state.

3. checkpoints
   - `denoiser.pt`: checkpoint for the denoising MLP
   - `classifier.pt`: checkpoint for the classifier MLP

6 Grading Policy

1. **Completeness**: 10 points for each Python file in "code" folder and report PDF file. 15 points for the report contents. If any plots are missing, marks will be deducted. (65 points)

2. **Quality of generation**: NLL score for unconditional generation: 10 points, for conditional generation: 5 pts for each state. In each of these cases, you'll get full marks only if you get an NLL score less than 2.5. Otherwise, you will not get points. The reason why we keep a hard cut is because if it's not less than 2.5 NLL, it means there is surely some bug in your program. In fact, without bugs, you should be able to get less than 2.3 NLL. ($10+5\times5 = 35$ points)

3. In total, 100 points.

7 Extra Reading

7.1 Nomenclature

Diffusion models are among the hottest topics in deep learning right now, mainly due to their hyperrealistic image and video generation capabilities. Due to this popularity, several terms related to diffusion models are thrown around blithely and can potentially confuse a new reader trying to understand the basics. We will go through some of the words commonly found in the generative model literature here.

The goal of **generative** modeling is to learn the underlying data generation process so that we may generate new samples that cannot be distinguished from the real samples. In deep learning, this "learning" is done by (1) choosing a statistical model and (2) learning the parameters for this model. This type of modeling is called **statistical** or **probabilistic** modeling. When this statistical model is built using neural networks, we call it **deep generative modeling**. Your choice of statistical model is crucial in probabilistic modeling. For example, if the data distribution that you aim to model is a mixture of Gaussians, but you choose to use a uni-modal Gaussian, it can lead to identifiability issues. However, in deep learning, people do not worry too much about this choice because we do not usually have prior information about the true underlying probabilistic model for the tasks where we expect deep learning to shine.

Most of the deep generative models share a belief that *the observed data (such as images) can be modeled using low-dimensional latent variables (sometimes called noise)*. The deep architectures vary in how they convert this noise to high-dimensional data. **Diffusion models** (DMs) [Soh+15] are one such architecture that models the stochastic process from a noise distribution to the high-dimensional data as a gas diffusion process. As an example, imagine the data distribution you want to model is the surface of a duck's body. You force the gas particles that have some temperature on the surface of the duck's body. Then, you place the duck in the center of a large vacuum cube. You start your clock and watch the gas particles slowly diffuse and occupy the entire volume of this cube. Diffusion models say that modeling the path of the gas particles from space to the duck's body will help us model the duck's body surface.

The term "diffusion model" refers to this method of leveraging a diffusion process to model a data generation process. It says nothing about how to learn DMs. Of course, in the paper, they have a method to learn a DM, but it is not the only method to learn a DM, and certainly not the way that popularized DMs. This is where **score-matching** comes in. Score-matching [Hyv05] was originally proposed to learn a general statistical model without having to learn the denominator[1] Song and Ermon [SE19] used score-matching to learn a diffusion model. This was concurrent with Ho, Jain, and Abbeel [HJA20], who also further simplified score matching as equivalent to denoising after every time step. Remember, these were not the first works to use denoising or score-matching to learn a generative model. They give due credit in their "related works" sections to those who attempted these methods before them. Nonetheless, they provided a stable method to train very powerful diffusion models.

DMs model the generative process similar to a thermodynamic process with stochasticity. These thermodynamic processes with stochasticity are modeled as **stochastic differential equations** (SDEs).

[1]Learning the denominator is one of the primary challenges in learning complex statistical models. This is called "intractability".

SDEs are similar to **ordinary differential equations** (ODEs) that we study in high school. The difference is that, in ODEs, we assume that we know/observe all the variables in the system, while in SDE, we model the unknown/unobservable variables as stochastic processes. **Flow-based models** [Lip+23] use ODEs instead of SDEs to model the generative process. See chapter 2 in the lecture notes of [HE25].

References

[Hyv05] Aapo Hyvärinen. "Estimation of Non-Normalized Statistical Models by Score Matching". In: *Journal of Machine Learning Research* 6.4 (2005).

[Soh+15] Jascha Sohl-Dickstein et al. "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". In: *International Conference on Machine Learning*. 2015.

[SE19] Yang Song and Stefano Ermon. "Generative Modeling by Estimating Gradients of the Data Distribution". In: *Advances in Neural Information Processing Systems*. 2019.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models". In: *Advances in Neural Information Processing Systems*. 2020.

[Son+21] Yang Song et al. "Score-Based Generative Modeling through Stochastic Differential Equations". In: *International Conference on Learning Representations*. 2021.

[Lip+23] Yaron Lipman et al. "Flow Matching for Generative Modeling". In: *International Conference on Learning Representations*. 2023.

[HE25] Peter Holderrieth and Ezra Erives. "Introduction to Flow Matching and Diffusion Models", MIT Course. 2025. URL: https://diffusion.csail.mit.edu/.