

EE 8520 - Homework 3

Unconditional Generation via Diffusion/Score-Based Models and Conditional Flow Matching

Due: November 7, 2025

Template: Any L^AT_EX template is fine

! Please follow the “rules for every HW” listed in the syllabus

Programming Exercise: In this assignment, you will implement several different state-of-the-art generative models (DDPMs/DDIMs, SDEs/PF-ODEs, and CFMs). You are provided with skeleton codes for each to help you get started.



Figure 1: Sample images from the Flickr-Faces-HQ (FFHQ) dataset (Karras et al., 2019), a high-quality collection of human face images originally introduced as a benchmark for evaluating generative adversarial networks (GANs).

Step 1 (45 points): Diffusion/Score-Based Models (DMs/SBMs)

Your first task is to sample from a pre-trained unconditional diffusion model called the Ablated Diffusion Model (ADM)¹ (Dhariwal & Nichol, 2021) developed by OpenAI. This model, along with its latent counterparts (EDM (Karras et al., 2022) and its successor EDM2 (Karras et al., 2024) developed by NVIDIA) represents the current state of the art in unconditional image generation. In recent literature on image synthesis, these models are consistently used as the leading benchmarks for diffusion-based generation and reconstruction.

To get you started, we have provided the skeleton code for sampling (`hw3_step1_main.py`) along with the ADM model codes and some helper functions (`utils.py`) under the folder `step1_utils`. Make sure to check the lines with the `#TODO` flag inside the `hw3_step1_main.py` file in order to complete the code. Some pointers about the model and code:

† We have provided the FFHQ pretrained ADM checkpoint ([click](#)). Please download this `FFHQ_ADM.pt` file and place it under `"./step1_utils/models/"`.

¹<https://github.com/openai/guided-diffusion>

- † As discussed in Lecture 11 and 12, recall that diffusion/score-based models can be parameterized in several equivalent ways, such as $\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t)$, $\hat{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t)$ or $\hat{\mathbf{s}}_\theta(\mathbf{x}_t, t)$. Remember that each predicts, respectively, the clean image, the added noise, or the score function. Despite their different formulations, these parameterizations are mathematically consistent and ultimately lead to the same generative process. The given pre-trained model was trained with the DDPM (Ho et al., 2020) objective to estimate the source noise $\boldsymbol{\epsilon}_0$ from \mathbf{x}_t (which is the common practice as mentioned in the class); therefore, the appropriate notation for this model is $\hat{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t)$.
- † As you know, these models use random Gaussian noise as the input during sampling. However, for the sake of reproducibility and fair comparison across questions, fixed random seeds have been defined in the `get_noise_x_t()` function for generating either 1 or 10 images. If you would like to produce completely random images for exploration or fun, you can change `--total_instances` to any value other than 1 or 10. Otherwise, please follow the specific values requested in each question to ensure consistent results.
- † We have given you a function called `extract_and_expand()` which takes a 1-D array (like a schedule of values for all timesteps) and extracts the element(s) corresponding to the current timestep(s) time, then reshapes and broadcasts it to match the shape of a target tensor. Make sure to use it before multiplying a coefficient (*e.g.*, A_t or B_t in Eq. 2) with a noisy/denoised latent (\mathbf{x}_t , $\hat{\mathbf{x}}_{0|t}$) or model outputs ($\hat{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t)$).
- † There are some parser operations that we have defined which you should change based on the question throughout the HW:
 - ◊ `--total_instances`: Specifies the number of image samples to generate.
 - ◊ `--diff_timesteps`: Indicates the total number of diffusion (denoising) steps used during model training. Keep this fixed at 1000.
 - ◊ `--desired_timesteps`: Defines the number of denoising steps to be used during sampling. You will modify this value from 1000 in questions (e) and (f).
 - ◊ `--eta`: Represents the η parameter in the DDIM sampling algorithm. You will experiment with this parameter in question (f).
 - ◊ `--schedule`: Specifies the sampling schedule, which can be either uniform or non-uniform. More details will be provided in question (e). **Ensure that the value or the sum of values specified here matches --desired_timesteps.** For example, if you are using 1000 denoising steps, set this to "1000". For 50 denoising steps, this could be a uniform split like "50", "25,25", "10,10,10,10,10" (which are all equivalent to "50"), or a non-uniform split such as "40,10", "30,20" or "30,15,5" (each representing a different distribution of timesteps).

For this step, your goals are:

- (a) [3 pts] Recall that in the DDPM, the forward diffusion process is defined as:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon} \quad (1)$$

where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$. Explain the roles of α_t , $\bar{\alpha}_t$, and β_t in this process, and clarify what each term controls. Describe how $\bar{\alpha}_t$ evolves as t increases (does it increase or decrease?), and discuss what this behavior reveals about the progression of the diffusion process.

(b) [8 pts] Below is the sampling algorithm for DDPMs.

Algorithm 1 Sampling for DDPM

```

1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
2: for  $t = T, \dots, 1$  do
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  if  $t > 1$ , else  $\mathbf{z} = \mathbf{0}$ 
4:    $\mathbf{x}_{t-1} \leftarrow \boldsymbol{\mu}_\theta(\mathbf{x}_t, t) + \sigma_t \mathbf{z}$ 
5: end for
6: return  $\mathbf{x}_0$ 
```

Look at the lecture slides to find the posterior mean approximation, $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$, for the $\hat{\epsilon}_\theta(\mathbf{x}_t, t)$ parametrization. Specifically, it can be represented using:

$$\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = A_t \cdot \mathbf{x}_t + B_t \cdot \hat{\epsilon}_\theta(\mathbf{x}_t, t) \quad (2)$$

where A_t and B_t are scalar coefficients that depend on α_t and $\bar{\alpha}_t$.

- Find A_t and B_t , and implement `sample_ddpm()` function given in `hw3_step1_main.py` which effectively performs the one-step denoising shown in lines 3 and 4 (Alg. 1).
- Run your DDPM sampling algorithm for 10 instances (random Gaussian noise inputs –use the given seeds) and for 1000 sampling steps. Report the sampling time per image and your generation results in a 2×5 grid, **comment** on them.
- Set `--total_instances = 1` and show the denoising process. Specifically, report $\mathbf{x}_{999}, \mathbf{x}_{750}, \mathbf{x}_{500}, \mathbf{x}_{250}, \mathbf{x}_{50}$, and \mathbf{x}_0 in a single row.

! Critical Note: For numerical stability and to ensure positivity, in most DDPM implementations, we work with the log-variance instead of the variance directly. Therefore, when we run the model:

```
model_output, model_var_values = torch.split(model_output, x_t.shape[1], dim=1)
model_log_variance = self.get_variance(model_var_values, t)
```

it should be noted that `model_log_variance = log(σ_t^2)`. Therefore, to calculate the correct σ_t , you should first perform:

$$\sigma_t = \sqrt{\sigma_t^2} = \exp\left(\frac{1}{2} \log(\sigma_t^2)\right) = \text{torch.exp}\left(0.5 \times \text{model_log_variance}\right)$$

Also please pay close attention to the **if condition** in line 3 (Alg. 1). Note that \mathbf{z} should also change for every t .

- (c) [8 pts] Refer to Eq. 1. In this equation, the true source noise is denoted by ϵ . However, in practice we do not have access to the true noise; instead, we estimate it using the model prediction $\hat{\epsilon}_\theta(\mathbf{x}_t, t)$. To obtain a denoised estimate of the clean image, substitute $\hat{\epsilon}_\theta(\mathbf{x}_t, t)$ in place of ϵ in Eq. 1 and express the result in the following form:

$$\hat{\mathbf{x}}_{0|t} = C_t \cdot \mathbf{x}_t + D_t \cdot \hat{\epsilon}_\theta(\mathbf{x}_t, t) \quad (3)$$

where C_t and D_t are scalar coefficients that depend on $\bar{\alpha}_t$.

- Derive C_t and D_t , and implement the `predict_x0_hat()` function provided in `hw3_step1_main.py`.
- Re-run the DDPM algorithm from part (a) for a single image instance and visualize the denoised estimate results across timesteps by displaying $\hat{\mathbf{x}}_{0|999}$, $\hat{\mathbf{x}}_{0|750}$, $\hat{\mathbf{x}}_{0|500}$, $\hat{\mathbf{x}}_{0|250}$, $\hat{\mathbf{x}}_{0|50}$, and $\hat{\mathbf{x}}_{0|0}$ in a single row. **Share your thoughts.**
- Note that this $\hat{\mathbf{x}}_{0|t}$ is also referred to as *Tweedie denoised estimate* in literature. Explain why by drawing the connection between SBMs and DMs. *Hint: Start from Tweedie's formula.*

- (d) [8 pts] Now using Eqs. 2 and 3, derive the following posterior mean approximation (show your derivation in detail):

$$\mu_\theta(\mathbf{x}_t, t) = \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} \mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)}{1 - \bar{\alpha}_t} \hat{\mathbf{x}}_{0|t} \quad (4)$$

which effectively implements the $\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t)$ parametrization. Based on Eq. 4, re-implement your `sample_ddpm()` function. Again, run your DDPM sampling algorithm for 10 instances (random Gaussian noise inputs –use the given seeds) and for 1000 sampling steps. Report your generation results in a 2×5 grid, **comment** on them. Are you getting the same generation results as part (a)?

Note: Either implement this in a new function or keep the part (a) solution by commenting it out for grading reasons.

- (e) [10 pts] As noted before, the model is trained using 1000 sampling steps, following the convention in literature. At sampling time, however, we often want to use fewer steps for efficiency. Let's say that we want to use 25 denoising steps as illustrated below:

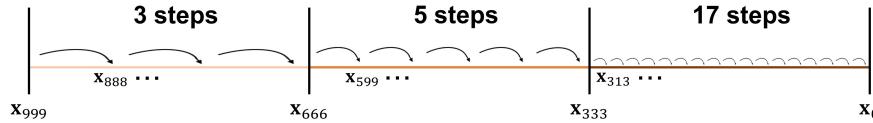


Now, the problem is that if we reduce the number of timesteps, we need to redefine a new sequence of β_t , α_t and $\bar{\alpha}_t$ that correspond to these new step indices while maintaining consistent noise levels. For example, your new β that describes the noise addition from $919 \rightarrow 959$ in the forward (noising) process would be:

$$\tilde{\beta} = 1 - \tilde{\alpha} = 1 - \frac{\bar{\alpha}_{959}}{\bar{\alpha}_{919}}$$

Note that because the reverse process is parameterized using that forward variance, you are going to be using *the same* β value, effectively implementing a fast schedule without retraining the model for 25 steps. Also note that you have access to $\bar{\alpha}_{959}$ and $\bar{\alpha}_{919}$, which you know from the precomputed cumulative products of α 's in your original diffusion schedule.

This can also be implemented using irregular jumps across the noise schedule. Specifically, think of the below example which uses 25 sampling steps again using the schedule "17,5,3" (first entry represents the segment):



Here, we effectively divide the noise schedule into 3 equal chunks but take different number of denoising steps. Specifically, we denoise the image more in the earlier stages and do more fine-tuned small denoising operations in the last steps.

- Implement the function `recreate_alphas()` which rebuilds a sampling noise schedule for a reduced set of timesteps. Use `utils.space_timesteps()` function (already given to you in the skeleton code) to obtain the kept original indices.
- Run your DDPM sampling algorithm for 10 instances (random Gaussian noise inputs –use the given seeds) and for **250** sampling steps using the schedule "250" (uniform jumps across schedule). Report the sampling time per image and your generation results in a 2×5 grid, **comment** on them and compare them to the 1000 sampling step results from part (a).
- Investigate the irregular (complex) noise schedules by setting `--schedule = "90,60,60,20,20"` which again uses 250 sampling steps. Report the sampling time per image and your generation results in a 2×5 grid, **comment** on them and compare them to uniform schedule generations.

- (f) [8 pts] We have seen in class that DDIM ([Song et al., 2021a](#)) defines an alternative non-Markovian inference process that shares the same marginal distributions as DDPM but introduces a deterministic mapping between noise and data samples. Specifically, we have seen that in DDIM sampling, the mean update is expressed as:

$$\mu_\theta(\mathbf{x}_t, t) = \sqrt{\bar{\alpha}_{t-1}} \hat{\mathbf{x}}_{0|t} + \sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} \hat{\epsilon}_\theta(\mathbf{x}_t, t)$$

Note that for DDIM, we have an explicit σ_t definition $\rightarrow \sigma_t = \eta \sqrt{\frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}} \sqrt{1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}}.$

- Discuss the role of η parameter. What does $\eta = 0.0$ and $\eta = 1.0$ mean?
- Implement `sample_ddim()` function given in `hw3_step1_main.py`. Run your DDIM sampling algorithm for 10 instances (random Gaussian noise inputs –use the given seeds) and for 1000 sampling steps using $\eta = 0.0$. Report your generation results in a 2×5 grid, **comment** on them. *Hint: Same DDPM sampling algorithm given in Alg. 1 also applies to DDIM sampling.*

- Repeat the previous item for $\eta = 1.0$. Do you remember seeing the same results anywhere in the previous questions?
- Now, set `--total_instances = 1` and `--eta = 0.0`. Report your generation results for a *single instance* (*keep the same single instance –same Gaussian noise input*) using 10, 20, 50, 100, and 1000 steps. What do you see? Please **comment**.
- Finally, again for a *single instance* (*keep the same single instance –same Gaussian noise input*), report your DDIM generation results (100 sampling steps) for $\eta \in \{0.0, 0.2, 0.5, 1.0\}$. What do you see? Please **comment**.

Step 2 (15 points): Score-SDEs and Probability Flow (PF) ODEs

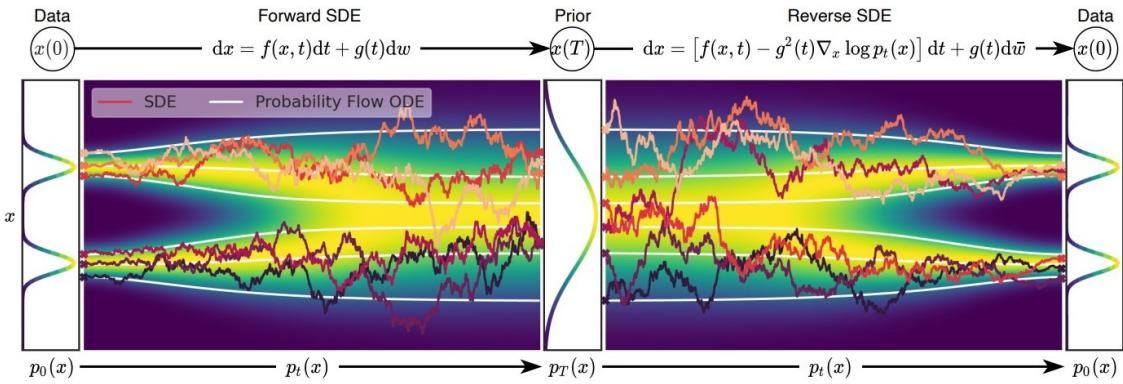


Figure 2: Forward and reverse processes of the toy model: score-based SDE and its deterministic version PF-ODE.

In this step, you will implement a toy illustration for both the continuous-time stochastic process (score-based SDE) and its deterministic counterpart (PF-ODE), producing a figure similar to Fig. 2. Note that for this toy example (mixture of Gaussian's), we can calculate the analytical form of $\nabla_{x_t} \log p_t(x)$. However, in practice, we estimate this using a neural network. You are given a skeleton code `hw3_step2_main.py`. Make sure to check lines with the `#TODO` flag in order to complete the code. For this step, your goals are:

- [8 pts] Implement the `forward_sde()` and `reverse_sde()` functions using the canonical formulas from Song et al. (2021b) for the Variance-Preserving SDE (VP-SDE), as also shown in Fig. 2. Hint: Be careful about the signs. The forward and backward integrations use the same `dt` magnitude, but `dt` integrates in opposite directions (forward vs. reverse), so the drift terms must be handled accordingly.
- [7 pts] Create new functions called `forward_ode()` and `reverse_ode()`, and implement them using the corresponding Probability-Flow ODE canonical formulas. These follow the same structure as the SDE versions but without the stochastic (noise) term. Hint: Canonical formulas for PF-ODEs differ from the ones you used implementing SDEs so please check the lecture slides for the correct formulation. Also note that similar sign considerations for `dt` apply.

Step 3 (15 points): Conditional Flow Matching (CFM)

Your final task is to implement a CFM (Lipman et al., 2023) model. The goal is to train a model that learns a time-dependent vector field which progressively transforms a noise sample into a data sample through a continuous flow. To keep the training manageable, you will train the model on MNIST. You are given skeleton codes `hw3_step3_main.py` and `step3_utils/cfm_model.py`. Make sure to check lines with the `#TODO` flag inside the `hw3_step3_main.py` file in order to complete the code.

In CFM (Lipman et al., 2023), the goal is to train a model that learns a time-dependent velocity field $v_\theta(\mathbf{x}_t, t)$ which transports samples from an initial noise distribution $p_0 = \mathcal{N}(0, I)$ to the data distribution $q_{\text{data}}(\mathbf{x})$ through an ordinary differential equation:

$$\frac{d\mathbf{x}_t}{dt} = v_\theta(\mathbf{x}_t, t; \mathbf{z}).$$

Consider $\mathbf{z} = \mathbf{x}_1$ and $\mathbf{x}_0 \sim p_0$. For each data point $\mathbf{x}_1 \sim q_{\text{data}}(\mathbf{x})$, we define a conditional probability path:

$$p_t(\mathbf{x}|\mathbf{x}_1) = \mathcal{N}(\mathbf{x}; \mu_t(\mathbf{x}_1), \sigma_t^2 I),$$

where $\mu_t(\mathbf{x}_1) = t\mathbf{x}_1$ and $\sigma_t = t\sigma_{\min} + (1-t)$. When $\sigma_{\min} = 0$, the path ends exactly on the clean data manifold, while a small $\sigma_{\min} > 0$ retains a slight amount of noise at $t = 1$ for improved numerical stability. In this homework, please set $\sigma_{\min} = 0$.

The network is trained to match the conditional velocity field by minimizing the flow matching loss:

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \mathcal{U}[0,1], \mathbf{x}_1 \sim q_{\text{data}}, \mathbf{x}_t \sim p_t(\cdot|\mathbf{x}_1)} \left[\|v_\theta(\mathbf{x}_t, t) - v(\mathbf{x}_t, t; \mathbf{x}_1)\|^2\right].$$

After training, new samples are generated by integrating the learned ODE forward in time from $t = 0$ to $t = 1$.

For this step, your goals are:

- (a) [3 pts] Implement the `compute_mu_t()` and `compute_sigma_t()` functions which computes the mean $\mu_t(\mathbf{x}_1)$ and the standard deviation σ_t .
- (b) [4 pts] Implement the `compute_conditional_velocity_field()` function using the formula for conditional vector field, $v(\mathbf{x}_t, t; \mathbf{x}_1)$. *Hint: Check the lecture slides for the conditional vector field formula.*
- (c) [3 pts] Implement the `sample_xt()` function, which draws a sample from the probability path $\mathcal{N}(\mathbf{x}; \mu_t(\mathbf{x}_1), \sigma_t^2 \mathbf{I})$ and define the loss function ($\mathcal{L}_{\text{CFM}}(\theta)$).
- (d) [5 pts] Train your CFM model on MNIST dataset. After training, generate new digit samples by solving the learned ODE. The sampling code is already provided in the skeleton files; however, you need to select your preferred ODE solver (refer to the [torchdiffeq documentation](#) for available options). Visualize the learning process by plotting the loss across epochs. **Comment** on the model performance.

References

- Prafulla Dhariwal and Alexander Nichol. Diffusion models beat GANs on image synthesis. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pp. 8780–8794, 2021.
- Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pp. 6840–6851, 2020.
- Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recog. (CVPR)*, pp. 4401–4410, 2019.
- Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2022.
- Tero Karras, Miika Aittala, Jaakko Lehtinen, Janne Hellsten, Timo Aila, and Samuli Laine. Analyzing and improving the training dynamics of diffusion models. In *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recog. (CVPR)*, 2024.
- Yedid Lipman, Shubham Batra, and Ruslan Salakhutdinov. Conditional flow matching: A new approach to generative modeling. In *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2023.
- Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021a.
- Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021b.