BMI / CS 771 Fall 2025:
Homework Assignment 4

Nov 2025

1 Overview

We have recently witnessed very exciting advances in deep generative models for image generation. In this assignment, you will explore two closely related families of models at the center of this progress: (latent) diffusion models and flow matching (FM) models. Despite their conceptual similarities and theoretical connections, each offers distinct perspectives on deep generative models. Specifically, you will implement (latent) **Denoising Diffusion Probabilistic Models (DDPM)** and **Rectified Flow** for *conditional image generation*, working with the MNIST and AFHQ datasets. While both DDPMs and Rectified Flow are built upon sophisticated mathematical foundations, you will find their implementation refreshingly approachable.

This assignment is team-based and requires cloud computing. A team is expected to have 2-3 students, unless otherwise approved by the instructor. The assignment has a total of 12 points with up to 12 bonus points. Details and rubric are described in Section 4.

2 Setup

- We recommend using Conda to manage your packages.
- The following packages are needed: PyTorch ($\geq$2.0 with GPU support), torchvision, PyYaml, NumPy, tqdm, and Tensorboard. Again, you are in charge of installing them.
- You can install any common packages that are helpful for the implementation. If a new package is required for your implementation, a description must be included in the writeup.
- You can debug your code and run experiments on CPUs. However, training a neural network is very expensive on CPUs. GPU computing is thus required for this project. Please setup your team's cloud instance. **Do remember to shut down the instance when it is not used!**

- Our helper code will automatically download MNIST dataset. We have also provided a script to download AFHQ (v2) [4]. Simply run *sh ./download_dataset.sh*

- To complete the assignment, you will need to fill in the missing code in: *./code/libs/ddpm.py*, *./code/libs/fm.py* and *./code/libs/unet.py*

- You are allowed to modify any part of the code in order to improve the model design or the training / inference of the model, though modification to other part of the helper code (besides *ddmp.py*, *fm.py*, and *unet.py*) is not be needed. Similar to HW3, you can experiment with different model designs and training hyperparameters using a YAML config file.

- Your submission should include the code, results, a trained model (see Section 6) and a writeup. The submission can be generated using: *python ./zip_submission.py*

3 Overview of the Implementation

Our helper code and the implementation (located under ./code) combines many pieces you previously saw in HW2 and HW3. You are required to go through our helper code before attempting this assignment.

Denoising Diffusion Probabilistic Models (DDPMs) [6] presents a class of generative models that synthesize high-quality images by gradually transforming noise into structured patterns. This process involves a series of steps where a model (often a deep model such as UNet [12]) learns to reverse a diffusion process, which adds noise to an image. DDPMs simulate a process where the model "denoises" an image, moving from random noise to a coherent, detailed image over multiple steps. In doing so, it is capable of mapping a sample from a white Gaussian noise into a realistic image.

Flow Matching (FM) [9] is a recent generative modeling framework that is conceptually similar to diffusion models. Like diffusion-based approaches, FM trains a neural network to approximate a vector field that defines a continuous transformation from a simple prior distribution to the target data distribution. Compared to DDPMs, FM is simpler to implement, converges more quickly, and typically requires far fewer inference steps [5]. These advantages have driven its rapid adoption in the research community.

Conditional Generation. Both DDPMs and FM were initially developed for unconditional generation, yet have since been adopted for text-conditioned generation (e.g., in Stable Diffusion [11]). In this assignment, we will consider generating images of digits or animal faces conditioned on their labels, i.e., inputting "1" and generating an image of "1".

Code Organization. Similar to HW3, the helper code has three parts.

- Source code for datasets, model architecture, DDPM, and training utilities are located under ./code/libs/.
- Configuration files (in yaml format) for the model architecture, training, and inference are stored under ./code/configs/.
- Training and evaluation interfaces are exposed in ./code/train.py.

DDMP Implementation (./code/libs/ddpm.py). This file implements a DDPM as in [6] and a latent diffusion model as in [11], including the forward diffusion process (q_sample), the reverse denoising process (p_sample and generate), as well as the learning of the denoising function (compute_loss). The current implementation misses several critical pieces in order to function properly. **You will need to complete the code here (p_sample, generate, q_sample, and compute_loss).** A good reference can be found in this link `https://huggingface.co/blog/annotated-diffusion`.

FM (Rectified Flow) Implementation (./code/libs/fm.py). This file implements rectified flow, a type of Flow Matching (FM) model, as in [9]. This implementation also supports flow matching in a latent space, similar to latent DDPMs. The current code misses several critical pieces in order to function properly. **You will need to complete the code here (generate and compute_loss).** A brief tutorial is included in Section 4.3. See also this link `https://diffusionflow.github.io/` for further details.

UNet Implementation (./code/libs/blocks.py and ./code/libs/unet.py). Central to DDPM and FM is the learning of a single step forward prediction function (e.g., the denoising function in DDPMs). Practically, this function is often implemented using a UNet [12]. Our helper code includes building blocks for UNet (./code/libs/blocks.py) and a partial implementation of UNet (./code/libs/unet.py). Our implementation, largely following the UNet used in Stable Diffusion [11], combines convolutional residual blocks, self-attention blocks, and cross-attention blocks to allow conditional image generation. **You will need to complete the implementation.**

Autoencoder Implementation (/code/libs/tiny_autoencoder.py). We have also included an open source implementation of a lightweight autoencoder [2], distilled from Stable Diffusion models [11]. The pre-trained weights are stored under /pretrained and referenced in the AFHQ config file. This autoencoder will be used for the implementation of latent DDPM and FM [11].

Training and Other Utilities (all other files under ./code/libs). These files implement utility functions that are necessary for the assignment. You can find a similar centralized parameter configuration module as in HW3 (*./libs/config.py*). Thus, you can use an external configuration file to control model architectures and training hyperparameters (in YAML format as *./code/configs/mnist_*.yaml* and *./code/configs/afhq_*.yaml*). Our helper code also supports common datasets (*./libs/datasets.py*) and metrics (FID score) for your exploration. *Most likely, you won't need to modify these files, even when attempting the bonus points.*

Training and Evaluation Interface. Our helper code provides a training interface as described below (assuming ./code as the current directory).

- To train a DDPM/FM model on MNIST dataset, run
  python ./train.py ./configs/mnist_[ddpm/fm].yaml

- To train a DDPM/FM model on AFHQ(v2) dataset, run
  python ./train.py ./configs/afhq_[ddpm/fm].yaml

- By default, the training logs and checkpoints will be saved under a folder in *../logs*. Each run will have a separate folder that ends with the starting time of the experiment. You can monitor and visualize these training logs using TensorBoard. Similar to our previous assignment, we recommend copying the logs and plotting the curves locally.
  tensorboard --logdir=../logs/your_exp_folder

- Our training script will generate new images using the current model when saving a new checkpoint. The checkpointing interval can be specified using "-c". For example, "-c 1" will create a checkpoints and generate new images at the end of every epoch. The resulting images (sample-#epoch.png) will be saved in the same folder as the checkpoints. A visual inspection of these images can help to monitor training progress.

- To evaluate a trained DDPM/FM model on AFHQ(v2) dataset, run
  python ./eval.py ./configs/afhq_[ddpm/fm].yaml path_to_checkpoint

  This evaluation will generate 1.5k samples from the trained model and store them in a subfolder within the experiment directory.

- You can resume from a previous checkpoint by using
  python ./train.py ./configs/mnist.yaml --resume path_to_checkpoint

4 Details

This assignment has three parts. An autograder will be used to grade certain parts of the assignment. **Please follow the instructions closely.**

4.1 UNet

Central to the implementation is a single step prediction function $f_\theta(x, c, t)$, realized using a UNet. In the vanilla version of DDPM, $f_\theta(x, c, t)$ takes the input of (a) a noisy image (x) at time step t of the diffusion process, and (b) a condition of image labels (i.e., context c), and predicts the noise. The estimated noise is then removed to produce a less noisy version of the image at time $t-1$. In FM with Rectified Flow, $f_\theta(x, c, t)$ also receives the input of (a) a noisy image (x) at time step t, and (b) a condition of image labels (i.e., context c), but instead predicts a vector field that specifies the instantaneous velocity toward the clean image along a continuous trajectory.

Understanding UNet (1 pt). We will start by reviewing the implementation of UNet. **You are expected to address the following questions.**

- Our implementation normalizes the input to the U-Net to be in the range of $[-1, 1]$ (see *./libs/datasets.py*). Can you explain why this is needed?
- How does the U-Net implementation handle the input time t and condition c (image labels)? Please describe the operations that inject the t and c into feature maps.

Completing the implementation of UNet (1 pt). Next, we will move forward and complete the implementation in *./libs/unet.py*. The missing code involves the forward pass of the UNet decoder. **Please complete the code and briefly describe your implementation in the writeup.**

4.2 DDPM and Latent DDPM

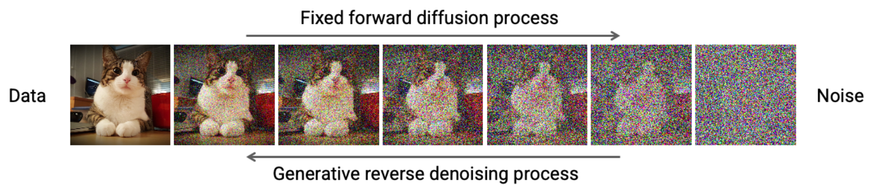
Fixed forward diffusion process
Data
Noise
Generative reverse denoising process

Figure 1: **Illustration of DDPM**. DDPM consists of (a) a forward diffusion process that gradually adds Gaussian noise to an input image, resulting in white Gaussian noises at the end of the process; and (b) a reverse denoising process that learns to remove the noise from a noisy version of the image, allowing for transforming a white Gaussian into a target data distribution.

Leveraging a denoising function, DDPM builds a reverse diffusion process to map an input simple distribution (Gaussian) to a target data distribution, as illustrated in Figure 1. We refer the technical details to the DDPM paper [6]. Additional technical discussion can be found in this excellent note [3] by Prof. Stanley Chan at Purdue University. **You will need to complete the following functions in** *./libs/ddpm.py* **and briefly describe your implementation in your writeup**.

- **Forward diffusion process (1 pt)**. Forward diffusion process is implemented in the *q_sample* function.
- **Reverse denoising process (2 pts)**. Reverse denoising process is implemented in *p_sample* (single step) and *generate* (full chain) functions.
- **Simplified DDPM loss (1 pts)**. This is implemented in *compute_loss* (see details in Algorithm 1 of [6]).

Experimenting with DDPM on the MNIST dataset (1 pt). Putting things together, we are now ready to train DDPMs on MNIST. With our default setup (30 epochs), the training may take an hour or two on a single T4 GPU. Digits should start to emerge in the sampled images before 10th epoch, and sampled images should begin to look fairly reasonable at the end of training. **In your writeup, please include (a) training curves; (b) samples from your model; and (c) a brief discussion of the results.**

Latent Diffusion Models (LDMs) (1 pt). Training DDPMs for high-resolution image generation is computationally expensive because the latent representation has the same spatial resolution as the output image, forcing the denoising function to operate directly on large inputs. A practical remedy is to use Latent Diffusion Models (LDMs) [11], where the diffusion process is carried out in a lower-resolution latent space rather than in pixel space. In LDMs, a learned encoder converts images into compact feature representations, and a DDPM is trained to model these latent features, substantially reducing the computational burden. The generated latent representations are then mapped back to the full-resolution image domain using a learned decoder. We refer readers to [11] for additional technical details. Our implementation of LDMs involves (1) the autoencoder, whose code and pre-trained weights are included; and (2) the DDPM, which you have already implemented. In the DDPM class, the use_vae variable controls the use of autoencoder, and thus switches between the vanilla DDPM and LDM. **You will need to implement the LDM using our help code**, which should be fairly straightforward with a few minor modifications to your DDPM implementation.

Experimenting with LDM on the AFHQ dataset (1 pt). We will train and evaluate LDMs on the high-resolution AFHQ dataset. The AFHQ training set contains 15,000 high-quality animal-face images at a resolution of 512×512, with an additional 1,500 images for testing. Full-resolution training on this dataset, however, requires substantial time and GPU resources. To make the assignment manageable, we adopt a simplified setting: images will be downsampled to 128×128 resolution, and training will be limited to 300 epochs. You will evaluate your trained model using FID. Under the default configuration, training typically takes a few hours on a single T4 GPU. Note that the resulting image quality will be limited due to the reduced training budget and the use of a vanilla DDPM, but you may still enjoy inspecting some of the amusing animal faces it produces. *Given the longer training time on AFHQ, please first ensure that your DDPM implementation is fully functional on the MNIST dataset.* **In your writeup, please include (a) training curves; (b) samples from your model; (c) FID scores of the final model; and (4)a brief discussion of these results.**

4.3 FM and Rectified Flow

You might have noticed that evaluating a trained DDPM model can take a fairly long time, as sampling from DDPMs requires many steps and has a high computational cost. To make the training more efficient, let us examine the core ingredients of DDPMs. As we discussed in the lecture, by scaling the time step t into the range of $[0, 1]$, a diffusion process can be re-written as

$$\begin{aligned} \text{(forward) diffusion kernel:} \quad & x_t = a_t x_0 + b_t \epsilon, \\ \text{(reverse) single-step denoising:} \quad & x_{t-\Delta t} = \hat{a}_t f_\theta(x_t, t) + \hat{b}_t \epsilon, \\ \text{where} \quad & x_0 \sim \text{[clean images]}, \quad x_{t\to 1} \sim \mathcal{N}(0, I) \\ & \epsilon \sim \mathcal{N}(0, I), \quad t \in [0,1], \quad \Delta t \to 0. \end{aligned} \tag{1}$$

We can simplify this process as follows. **First**, select $a_t = 1-t$ and $b_t = t$, which ensures $x_{t\to 1} = \epsilon \sim \mathcal{N}(0, I)$. **Second**, drop the stochasticity in the reverse process by setting $\hat{b}_t = 0$. **Finally**, re-parameterize $\hat{a}_t f_\theta$ as $x_t - \Delta t f_\theta(x_t, t)$, yielding an update rule that closely resembles the denoising step used in DDPMs.

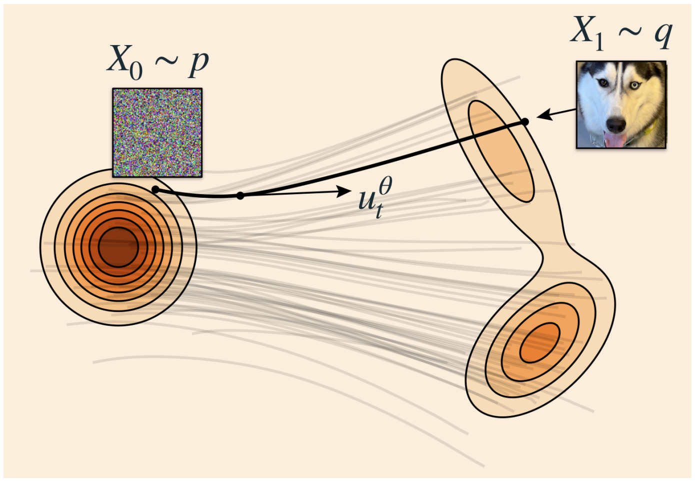

Figure 2: **Illustration of FM**. FM with rectified flow learns a time-dependent vector field that gradually transports samples from a prior distribution (e.g., Gaussian noise) to samples from the target distribution (e.g., natural images).

$$\begin{aligned} \text{(forward) diffusion kernel:} \quad & x_t = (1-t)\ x_0 + t\ x_{t\to 1}, \\ \text{(reverse) single-step denoising:} \quad & x_{t-\Delta t} = x_t - \Delta t\ f_\theta(x_t, t), \\ \text{where} \quad & x_0 \sim \text{[clean images]}, \quad x_{t\to 1} \sim \mathcal{N}(0, I) \\ & t \in [0,1], \quad \Delta t \to 0. \end{aligned} \tag{2}$$

To align our formulation with common notation in the literature, we further reverse the time parameter using $\tau = 1 - t$. This change of variables yields the rectified flow model [9]

$$\begin{aligned}
\text{forward:}\quad & x_\tau = \tau\ x_{\tau=1} + (1-\tau)\ x_{\tau=0}, \\
\text{reverse (single step):}\quad & x_{\tau+\Delta\tau} = x_\tau + \Delta\tau\ f_\theta(x_\tau, \tau), \\
\text{where}\quad & x_{\tau=1} \sim [\text{clean images}],\quad x_{\tau=0} \sim \mathcal{N}(0, I) \\
& \tau \in [0,1],\quad \Delta\tau \to 0.
\end{aligned} \tag{3}$$

Since $x_{\tau=0} \sim \mathcal{N}(0, I)$, sampling an image $x_{\tau=1}$ amounts to iteratively apply the single-step reverse update to trace a trajectory from a sampled $x_{\tau=0}$ to a clean image $x_{\tau=1}$. On the other hand, training amounts to learning f_θ to predict the instantaneous velocity $(dx_\tau/d\tau)$ along the trajectory between noise $x_{\tau=0}$ and an image $x_{\tau=1}$. In this simplified setting, the velocity field is constant and equal to $x_{\tau=1} - x_{\tau=0}$. Intuitively, Rectified Flow can be viewed as learning a time-dependent vector field that gradually transports samples from a prior distribution (e.g., Gaussian noise) toward samples from the target distribution (e.g., natural images). This process is illustrated in Figure 2.

While the Rectified Flow paper [9] provides extensive theoretical detail, it can be challenging to read on a first pass. Instead, we recommend starting with Sections 1 and 2 of the more accessible tutorial [8], which includes intuitive explanations and accompanying code examples. **You will need to implement both training and inference procedures in the corresponding functions in *./libs/fm.py* and briefly describe your implementation in the writeup**. Your implementation is expected to support Rectified Flow in a latent feature space, similar to LDMs.

- **Sampling process (1 pt)**. This is implemented in the *generate* function. The implementation is very similar to reverse denoising and indeed follows the forward Euler method.
- **Rectified Flow loss (1 pt)**. This is implemented in the *compute_loss* function (see details in Algorithm 1 of [9]). Again, the implementation is similar to DDPM's loss.

Experimenting with FM on the AFHQ dataset (1 pt). Again, we will train and evaluate FM on the AFHQ dataset, following the same setup used for LDMs. To help you verify and debug your FM implementation, we have provided a YAML config file for training FM on the MNIST dataset. *Please ensure that your model produces meaningful results on MNIST before moving on to the AFHQ experiments.* **In your writeup, please include (a) training curves; (b) samples from your model; (c) FID scores of the final model; and (4) a brief discussion of these results in comparison to those from DDPM.**

4.4 Bonus

This assignment offers several interesting directions for further exploration. To inspire creativity and encourage deeper engagement, we are offering bonus points for students who pursue these additional challenges.

Improving U-Net (+4 pts). A key component shared by both DDPMs and FMs is the U-Net architecture. Recent work has explored various U-Net refinements [7] and even replacing U-Net with vision Transformers [10]. Bonus points will be awarded for experimenting with improved U-Net design on the AFHQ dataset. We strongly recommend implementing your modifications in a separate U-Net class. **Please include your implementation, describe your solution, and discuss the results in the writeup**.

Improving DDPMs (+4 pts). DDPMs lead to a family of deep generative models. Many of these models share a similar denoising step, yet has different mathematical formulations and derivations. A recent work seeks to unified these models [7] under a simplified framework, demonstrating improved generation quality and faster training convergence. Bonus points will be provided for exploring improved versions of DDPMs on the AFHQ dataset, using [7] or other approaches. Again, a separate DDPM class is recommended for the implementation. **Please include your implementation, describe your solution, and discuss the results in the writeup**.

Scaling up the Diffusion Model (+4 pts). You might have already noticed that our implementation of UNet has a few non-standard parts, such as the MLP used in the Transformer block. These are meant for compatibility with Stable Diffusion models. In fact, with some minor modifications to our UNet, one should be able to load Stable Diffusion 1.x checkpoints [1]. Bonus points will be provided for loading Stable Diffusion checkpoints using your implementation and demonstrating successful inference. You will need a few things to make this happen: (1) a modified UNet (a new class is preferred); (2) a CLIP text encoder from OpenAI (also on Hugging Face); (3) a function that loads Stable Diffusion checkpoints and set the model weights in our code; and (4) a proper config file for LDM. **Please include your implementation, describe your solution, and discuss the results in the writeup**.

5 Writeup

For this assignment, and all other assignments, you must submit a project report in PDF. Every team member should send the same copy of the report. Please clearly identify the contributions of all members. For this assignment, you will need to include (a) responses to the questions; (b) descriptions of your implementation; (c) results on MNIST and AFHQ; and (d) discussions of your results. You can also discuss anything extra you did (e.g., for the bonus points). Feel free to add other information you feel is relevant.

6 Handing in

This is very important as you will lose points if you do not follow instructions. Every time after the first that you do not follow instructions, you will lose 5%. The folder you hand in must contain the following:

- code/ - directory containing all your code for this assignment
- writeup/ - directory containing your report for this assignment.
- results/ - directory containing additional results. Visuals including training curves and sampled images can be included in the writeup. This folder is intended for any results that could not fit into the report.

Do not use absolute paths in your code (e.g. /user/classes/proj1). Your code will break if you use absolute paths and you will lose points because of it. Simply use relative paths as the starter code already does. **Do not turn in the data / logs / models folder**. Hand in your project as a zip file through Canvas. You can create this zip file using *python zip_submission.py*.

References

[1] Stable diffusion. `https://github.com/CompVis/stable-diffusion`, 2022.

[2] Tiny autoencoder for stable diffusion. `https://github.com/madebyollin/taesd`, 2022.

[3] S. H. Chan. Tutorial on diffusion models for imaging and vision. *arXiv preprint arXiv:2403.18103*, 2024.

[4] Y. Choi, Y. Uh, J. Yoo, and J.-W. Ha. Stargan v2: Diverse image synthesis for multiple domains. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8188–8197, 2020.

[5] P. Esser, S. Kulal, A. Blattmann, R. Entezari, J. Müller, H. Saini, Y. Levi, D. Lorenz, A. Sauer, F. Boesel, et al. Scaling rectified flow transformers for high-resolution image synthesis. In *International conference on machine learning*, 2024.

[6] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[7] T. Karras, M. Aittala, T. Aila, and S. Laine. Elucidating the design space of diffusion-based generative models. *Advances in neural information processing systems*, 35:26565–26577, 2022.

[8] Y. Lipman, M. Havasi, P. Holderrieth, N. Shaul, M. Le, B. Karrer, R. T. Chen, D. Lopez-Paz, H. Ben-Hamu, and I. Gat. Flow matching guide and code. *arXiv preprint arXiv:2412.06264*, 2024.

[9] X. Liu, C. Gong, et al. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *The Eleventh International Conference on Learning Representations*, 2023.

[10] W. Peebles and S. Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4195–4205, 2023.

[11] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[12] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.