SFU

Practices in Visual Computing II (Spring 2025)
Assignment 3
Image Generation with Diffusion Model

Due: Friday, 14th March, 23:55 pm

Overview

In this assignment, you will be implementing **DDPM (Denoising Diffusion Probabilistic Models)**, a type of generative model used for image synthesis. Note that the DDPM described here is a simplified version. If there are any discrepancies between these instructions and the **DDPM paper**, please follow this assignment. You can still refer to the DDPM paper for further understanding.

1 Preliminary

Understanding the theory behind diffusion models significantly simplifies their implementation. To gain a solid foundation, it is highly recommended to watch the following YouTube video and review the DDPM paper before proceeding with the next sections. These resources will provide a clearer intuition about how diffusion models work.

- **YouTube Video:** How I Understand Diffusion Models by Jia-Bin Huang.
- **DDPM Paper:** Denoising Diffusion Probabilistic Models.

Forward Process

Denoising Diffusion Probabilistic Model (DDPM) is one of the latent-variable generative models consisting of a Markov chain. In this Markov chain, we define a *forward process* that gradually adds noise to data $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ so that $\mathbf{x}_0$ becomes pure white Gaussian noise at $t = T$. Each transition of the forward process is:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) := \mathcal{N}\Big(\mathbf{x}_t;\ \sqrt{1-\beta_t}\,\mathbf{x}_{t-1},\ \beta_t\mathbf{I}\Big),$$

where a variance schedule $\beta_1, \ldots, \beta_T$ controls the step sizes.
Because of the properties of Gaussian distributions, we can directly sample $\mathbf{x}_t$ at an arbitrary timestep t from real data $\mathbf{x}_0$ in closed form:

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\Big(\mathbf{x}_t;\ \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0,\ (1-\bar{\alpha}_t)\,\mathbf{I}\Big),$$

where $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$.

Reverse Process

If we can reverse the forward process, i.e., sample $\mathbf{x}_{t-1} \sim q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ iteratively until $t = 0$, we can generate $\mathbf{x}_0$ close to the true data distribution $q(\mathbf{x}_0)$ from Gaussian noise $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$. This *reverse process* is a denoising chain that gradually transforms noise into a real-looking sample.
The reverse process is also a Markov chain with learned Gaussian transitions:

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t),$$

where $p(\mathbf{x}_T) = \mathcal{N}(0, \mathbf{I})$ and

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) := \mathcal{N}\big(\mathbf{x}_{t-1};\ \boldsymbol{\mu}_\theta(\mathbf{x}_t, t),\ \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)\big).$$

Training

To learn this reverse process, we minimize the KL divergence between $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ and $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}\big(\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0),\ \sigma_t^2\mathbf{I}\big)$, which is also Gaussian when conditioned on $\mathbf{x}_0$:

$$\mathcal{L} = E_q\left[\sum_{t>1} D_{\mathrm{KL}}\big(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)\big)\right].$$

A standard parameterization sets $\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t) = \sigma_t^2\mathbf{I}$ as untrained time-dependent constants. Thus, the objective becomes:

$$\mathcal{L} = E_q\Big[\frac{1}{2\,\sigma_t^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t, t)\|^2\Big] + C.$$

Empirically, predicting the noise $\boldsymbol{\epsilon}$ injected into data (rather than directly predicting $\boldsymbol{\mu}$, which is the mean of the true reverse process) often yields better results. Therefore, in practice, we use a noise prediction network $\boldsymbol{\epsilon}_\theta$ and a simplified objective:

$$\mathcal{L}_{\text{simple}} := E_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\Big[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\big(\mathbf{x}_t(\mathbf{x}_0, t), t\big)\|^2\Big],$$

where

$$\mathbf{x}_t(\mathbf{x}_0, t) = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}).$$

Sampling

Once $\boldsymbol{\epsilon}_\theta$ is trained, we can sample from the model by gradually denoising white Gaussian noise. The DDPM sampling procedure iteratively transforms $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$ into a sample $\mathbf{x}_0$.

2 Workspace Initialization (0 Points)

We will use the **AFHQ (Animal Faces-HQ)** dataset, which contains 16,130 high-quality images at a resolution of 512×512. It has three classes (Cat, Dog, and Wildlife), each with around 5,000 images, offering a diverse collection across various breeds.
To begin training, simply run:

```
python training.py
```

You do not need to worry about handling or downloading the dataset, as the provided code does so automatically.

3 Scheduler (10 Points)

In a diffusion model, noise scheduling is critical for both the forward (adding noise) and reverse (denoising) processes. We need to compute:

- β (beta)
- $\alpha = 1 - \beta$
- $\hat{\alpha}_t = \prod_{s=1}^{t} \alpha_s$ (cumulative product of α)
- σ (sigma)

Defining β

The values of β specify how much noise is added at each step, and different definitions influence the overall noise growth schedule.
Linear Mode:

$$\beta_t = \beta_1 + \frac{t-1}{T-1}(\beta_T - \beta_1), \quad t = 1, \dots, T.$$

This defines β_t as increasing linearly from β_1 to β_T over T steps.
Quadratic Mode:

$$\beta_t = \left(\sqrt{\beta_1} + \frac{t-1}{T-1}(\sqrt{\beta_T} - \sqrt{\beta_1})\right)^2, \quad t = 1, \dots, T.$$

Here, β_t follows a quadratic schedule, starting with smaller increments and increasing more rapidly toward the end.

Defining α and $\hat{\alpha}$

Once β_t is set:

$$\alpha_t = 1 - \beta_t,$$

$$\hat{\alpha}_t = \prod_{s=1}^{t} \alpha_s.$$

α_t decreases as β_t increases, and the cumulative product $\hat{\alpha}_t$ shrinks with each step, indicating how much of the original signal remains.

Defining σ

σ_t controls the noise reintroduced during the reverse process. Two common definitions follow:

Small σ Approach:

$$\sigma_t = \sqrt{\frac{(1 - \hat{\alpha}_{t-1})}{(1 - \hat{\alpha}_t)}\,\beta_t}.$$

This keeps variance relatively low at each reverse step.

Large σ Approach:

$$\sigma_t = \sqrt{\beta_t}.$$

A larger variance can lead to more stochasticity in generation, potentially yielding more diverse samples.

4 Forward and Reverse Diffusion (10 Points)

In diffusion models, the forward process adds noise to data, and the reverse process removes it.

Add Noise (`add_noise`)

For a clean sample $\mathbf{x}_0$ at $t = 0$, to get a noisy version $\mathbf{x}_t$ at an arbitrary t:

$$\mathbf{x}_t = \sqrt{\hat{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1 - \hat{\alpha}_t}\,\boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and

$$\hat{\alpha}_t = \prod_{s=1}^{t} \alpha_s, \quad \alpha_s = 1 - \beta_s.$$

Since $\hat{\alpha}_t$ gets smaller with t, most of the original data is destroyed by large t.

Reverse Step (`step`)

Let $\mathbf{x}_t$ be the noisy sample at step t. Given a model-predicted noise $\boldsymbol{\epsilon}_\theta$, the mean for the reverse step is:

$$\boldsymbol{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \big(\tfrac{1-\alpha_t}{\sqrt{1-\hat{\alpha}_t}}\big)\boldsymbol{\epsilon}_\theta\Big).$$

We can sample $\mathbf{x}_{t-1}$ as:

$$\mathbf{x}_{t-1} = \begin{cases} \boldsymbol{\mu}_t + \sigma_t \boldsymbol{\eta}, & t > 0, \\ \boldsymbol{\mu}_t, & t = 0, \end{cases} \quad \boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).$$

Here, σ_t can be chosen as $\sqrt{\beta_t}$ (large) or via the smaller approach above.

5 U-Net Model (0 Points)

U-Net is central to denoising in many diffusion models. Although its encoder-decoder structure can optionally predict a denoised sample x_0, in practice, most diffusion formulations use U-Net to predict the added noise $\hat{\epsilon}(x_t, t)$.

5.1 Architecture and Forward Pass

Downsampling and Upsampling. U-Net uses a series of Down blocks to progressively reduce spatial resolution, capturing higher-level features. Each Down block applies convolution with stride 2, followed by normalization (e.g., GroupNorm) and a nonlinear activation (e.g., SiLU). In parallel, Up blocks restore the original resolution via nearest-neighbor interpolation and convolution, merging skip-connected features from the Down blocks to recover fine-grained details.

Residual Blocks and Self-Attention. Each stage contains Residual Blocks that can incorporate optional self-attention. Self-attention allows every spatial position to leverage global context, complementing local convolutional operations.

Time Embedding. A scalar timestep t is encoded by sinusoidal embeddings, then transformed by an MLP. The resulting time-dependent vector is injected into the Residual Blocks, conditioning the network on the current diffusion step.

Noise Prediction. At each diffusion step t, the noisy input x_t (along with the time embedding) passes through Down blocks, a bottleneck layer, and Up blocks. A final convolutional layer outputs $\hat{\epsilon}$, the predicted noise. This prediction is then used to iteratively denoise x_t, guiding it toward the original data distribution.

6 Residual Block (10 Points)

A **Residual Block** is a fundamental component in many CNN-based architectures, facilitating deeper networks by making gradient flow more efficient. See Table 1 for a detailed breakdown of the Residual Block structure, including input/output shapes and layer configurations.

Key Components

- **Normalization and Activation (GroupNorm + SiLU)**
- **Convolutional Layers (Conv2D)**
- **Dropout**
- **Time Embedding Injection**: Integrates temporal/context embedding.
- **Shortcut Connection (1×1 Conv / Identity)**: Adds input back to the output, enabling residual learning.

7 Time Embedding Block (5 Points)

The **Time Embedding** block maps a scalar timestep t to a higher-dimensional feature vector, combining sinusoidal encodings with an MLP.

Blocks	Layers	Dropout	Conv Params	Input Shape	Output Shape
Block 1	GroupNorm	-	-	$[B, C_i, H, W]$	$[B, C_i, H, W]$
	SiLU	-	-	$[B, C_i, H, W]$	$[B, C_i, H, W]$
	Conv2D	-	K=3, S=1, P=1	$[B, C_i, H, W]$	$[B, C_o, H, W]$
Time Inject	Add Time Emb	-	-	$[B, C_o, H, W]$	$[B, C_o, H, W]$
Block 2	GroupNorm	-	-	$[B, C_o, H, W]$	$[B, C_o, H, W]$
	SiLU	-	-	$[B, C_o, H, W]$	$[B, C_o, H, W]$
	Dropout	0.1	-	$[B, C_o, H, W]$	$[B, C_o, H, W]$
	Conv2D	-	K=3, S=1, P=1	$[B, C_o, H, W]$	$[B, C_o, H, W]$
Shortcut	1×1 Conv / ID	-	K=1, S=1, P=0	$[B, C_i, H, W]$	$[B, C_o, H, W]$

Table 1: Residual Block Structure with Input/Output Shapes. C_i = Input channels, C_o = Output channels, H = Height, W = Width. Conv Params: K = kernel size, S = stride, P = padding.

Key Components

- **Sinusoidal Embeddings (`sin_emb`)**: Inspired by positional encodings in Transformers. Each timestep t is represented with multiple frequencies:

$$\omega_i = \exp\left(-\frac{\log(\texttt{max\_period})}{n} \times i\right), \quad i \in [1, \ldots, n/2].$$

  The embedding becomes:

$$\text{emb}(t) = \left[\cos(t \cdot \omega_1),\ \sin(t \cdot \omega_1),\ \ldots,\ \cos(t \cdot \omega_{n/2}),\ \sin(t \cdot \omega_{n/2})\right].$$

- **Multi-Layer Perceptron (MLP)**: A two-layer network with a SiLU activation that transforms the sinusoidal features into a higher-dimensional representation.

Step	Layer / Operation	Parameters	Input Shape	Output Shape
Sinusoidal	sin_emb (cos,sin)	freq=256, max_period=10000	$[B]$ or $[B, 1]$	$[B, \texttt{freq}]$
MLP	Linear + SiLU + Linear	hidden_dim=H	$[B, \texttt{freq}]$	$[B, H]$

Table 2: Time Embedding Block Structure. The sinusoidal embedding first encodes the scalar t, and an MLP projects it to a hidden dimension H.

8 Diffusion Module (10 Points)

This module defines the **loss function** and **sampling function** in a typical diffusion model.

Loss Computation (`get_loss`)

- Randomly pick a timestep t from $[1, \dots, T]$.
- Noise the original sample $\mathbf{x}_0$ to $\mathbf{x}_t$:

$$\mathbf{x}_t = \sqrt{\hat{\alpha}_t}\,\mathbf{x}_0 \; + \; \sqrt{1-\hat{\alpha}_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).$$

- The model predicts $\hat{\boldsymbol{\epsilon}}$ from $\mathbf{x}_t$ and t, and we compute an MSE loss:

$$\mathcal{L} = \|\hat{\boldsymbol{\epsilon}} - \boldsymbol{\epsilon}\|^2.$$

Minimizing this encourages the model to accurately invert the forward (noising) process.

Sampling (`sample`)

To generate new samples, start with noise $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and apply the reverse `step` for each $t = T, T-1, \dots, 1$ until reaching $t = 0$. The final $\mathbf{x}_0$ is a generated image.

9 Evaluation (0 Points)

Evaluation involves generating sample images and computing the Frechet Inception Distance (FID) to measure image quality.

Sampling Images

```
python evaluate.py sample --ckpt_path <path_to_ckpt> --save_dir <save_dir>
```

Replace `<path_to_ckpt>` with your checkpoint path and `<save_dir>` with where to save results.

Calculating FID

```
python evaluate.py fid --save_path <path_to_GI> --gt_path <path_to_GT>
```

Here, `<path_to_GI>` is the directory with your generated images, and `<path_to_GT>` is the directory containing ground-truth images.
What is FID? FID compares the distribution of generated images with that of real images; lower values indicate higher similarity to real data.

10 Demo (15 Points)

At the demo session:

1. We will check if you implemented and can run the code properly.
2. Your model must produce reasonable results (An FID under 20).
3. You will need to explain part of your code during the demo.
4. We will test your understanding with a conceptual question.

Note: Submit your code, training logs, and 10 generated samples to Canvas before the deadline. You may not change them during the demo session.