CSCI-GA 2271: Assignment 3

Due: Thursday, November 27, 2025 at 6:00 PM.

1 Part A: Variational Autoencoder [100 pts]

Under A3_PartA navigate to `variational_autoencoders.ipynb` and read through the `ipynb` instructions on how to code up a Variational Autoencoder (VAE) from scratch. You are only required to modify `vae.py` in the specified lines and run the cells in the `.ipynb` file. Make sure all the helper files and folders are in the same directory to avoid hiccups. Submit your `.ipynb` file and `.py` file as a single zip file.

2 Part B(Optional): Diffusion Models Bonus [50 pts]

In this section, you will dive into the practical aspects of implementing diffusion models. Throughout this programming assignment, you will gain hands-on experience into state-of-the-art techniques for image generation and denoising tasks.

It's worth noting that, due to limited computing resources, the dataset provided for this exercise is only a subset of the original dataset. Therefore, the quality of the generated images may not meet expectations. Nevertheless, your experimentation with DDPM will offer valuable insights into its capabilities and potential for broader applications in machine learning and data analysis. Upon completion, you'll have acquired practical experience in building and leveraging DDPMs, opening doors to a deeper understanding of diffusion models.

Dataset

The dataset for this homework is the Animal Faces-HQ dataset (AFHQ), consisting of 15,000 high-quality images at 512×512 resolution. The dataset includes three domains of cat, dog, and wildlife, and in our assignment you **only need to use cat** images to reduce the computation complexity.

Starter Code

The main structure of the files is organized as follows:

```
A3_PartB/
   data/
   diffusion.py
   main.py
   requirements.txt
   run_diffusion.ipynb
   trainer.py
```

`unet.py`
`utils.py`

Here is what you will find in each file:

1. `data`: Contains the AFHQ dataset.
2. `diffusion.py`: Constructs the diffusion model, including the forward process, backward process, and scheduler, which you will implement. (Hint: This is the **only** file you need to modify. Locations in the code where changes ought to be made are marked with a **TODO**.)
3. `main.py`: Serves as the main entry point for training and evaluating your diffusion model IF you are running locally or on AWS. You won't need this file if you are running on Colab or Kaggle. Append flags to this command to adjust the diffusion model's configuration.
4. `requirements.txt`: Lists the packages that need to be installed for this assignment.
5. `run_diffusion.ipynb`: Provides command lines to train and evaluate your diffusion model in Google Colab or Kaggle.
6. `trainer.py`: Provides code for training and evaluating the diffusion model.
7. `unet.py`: Contains code for the U-Net network, which aims to model the denoising function for the diffusion model.
8. `utils.py`: Helper functions to simply the process of training or evaluating your diffusion model.

Parameters

In table 1,2 and 3, you will find all of the parameters which can be configured in the starter code. You can set these parameters in functions `train_diffusion` or `visualize_diffusion` as seen in `run_diffusion.py`.

Description	Parameter	Default Value
Directory from which to load data	| data_path|	(See starter notebook)
Number of iterations to train the model	| train_steps|	(See handout below)
Enable FID calculation	| fid|	(See handout below)
Frequency of periodic save, sample and (optionally) FID calculation	| save_and_sample_every|	(See handout below)

Table 1: Useful parameters for `run_diffusion.py`

Description	Parameter	Default Value
Dataloader worker threads	| dataloader_workers|	16
Directory where the model is stored	| save_folder|	| ./results/|
Path of a trained model	| load_path|	| ./results/model.pt|

Table 2: Additional parameters for `run_diffusion.py`. You likely won't need to change these

Description	Parameter	Default Value
Model image size	| image_size|	32
Model batch size	| batch_size |	32
Data domain of AFHQ dataset	| data_class|	| cat|
Number of steps of diffusion process, T	| time_steps|	50
Number of output channels of the first layer in U-Net	| unet_dim|	16
Learning rate in training	| learning_rate|	1e-3
U-Net architecture	| unet_dim_mults|	[1, 2, 4, 8]

Table 3: Additional parameters for `run_diffusion.ipynb`. These won't need to be changed from default values for this homework.

Google Colab

Colab provides a free T4 GPU for code execution, albeit with a time limitation that may result in slower training. In the event of GPU depletion on Colab, options include waiting for GPU recovery, switching Google accounts, purchasing additional GPU resources, switching to Kaggle, or switching to a cloud provider (such as GCP or AWS).

To download the AFHQ dataset on Colab, run the commands below. Ensure to prepend "!" before the commands below when working on Colab. If you mount your drive, you should only need to run this once. See the `run_diffusion.ipynb` file for more details

```
mkdir -p ./data
!gdown --id 1-1npXlqCw1CYQ5SBbrlSgZdA9TEwZre4 -O ./data/
   afhq_v2.zip
unzip -q ./data/afhq_v2.zip -d ./data
```

Kaggle

Kaggle provides 30 hours of free T4 or V100 GPU runtime per week. This is more than sufficient to complete this homework.

Diffusion

In this problem, you will implement Denoising Diffusion Probabilistic Models (DDPM) (Ho et al., 2020), in the `Diffusion` class in `diffusion.py`.

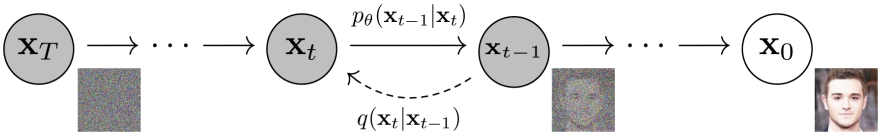

Figure 1: The Markov chain of forward (reverse) diffusion process of generating a sample by slowly adding (removing) noise.

Forward Process (Noise ← Image): In this problem, $\mathbf{x}_0 \sim q(\mathbf{x})$ corresponds to the pixels of the image. The *forward diffusion* process sequentially applies a small amount of Gaussian noise to the data sample $\mathbf{x}_0$ for T steps, producing a sequence of noisy samples $\mathbf{x}_1, \ldots, \mathbf{x}_T$.

The diffusion step can be derived as:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{\alpha_t}\mathbf{x}_{t-1}, 1 - \alpha_t \boldsymbol{I}), \tag{1}$$

where $\mathbf{x}_t$ is the image after t diffusion steps, $\boldsymbol{I}$ is the identity matrix. The step sizes are controlled by a variance schedule $\{\alpha_t \in (0,1)\}_{t=1}^T$ such that the data sample $\mathbf{x}_0$ gradually loses its distinguishable features as step t becomes larger. This is shown in Fig. 1.

Using the reparameterization trick, we can sample $\mathbf{x}_t$ directly from $\mathbf{x}_0$:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, \text{where } \boldsymbol{\epsilon} \sim \mathcal{N}(0, I). \tag{2}$$

Noise Schedule: In this assignment, we use the improved cosine-based variance schedule of (Nichol & Dhariwal, 2021):

$$\begin{aligned} \alpha_t = \text{clip}\left(\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, 0.001, 1\right), \bar{\alpha}_t = \frac{f(t)}{f(0)}, \\ \text{where } f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2, \end{aligned} \tag{3}$$

and we set $s = 0.008$ to prevent α_t from becoming too large when close to $t = 0$.

Reverse Process (Noise → Image): Ho et al. (2020) proved that the ELBO for a diffusion model can be rewritten as:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)] - \sum_{t=2}^{T} \underbrace{\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{KL}\left(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0), p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)\right)\right]}_{\mathcal{L}'_t} + C, \tag{4}$$

(see Appendix A in their paper linked above). The reverse model $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as shown in Fig. 1 is trained to maximize the lower bound of Equation 4. Note that the forward process $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ does not contain any trainable parameters.

The distributions $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ inside the $\mathcal{L}'_t$ terms of Equation 4 can act as a "ground-truth signal", since they define how to denoise a noisy image $\mathbf{x}_t$ with access to what the final,

completely denoised image $\mathbf{x}_0$ should be. Using the Markov properties of a diffusion model, it is possible to show that the distribution $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ decomposes as

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)}. \tag{5}$$

From this Equation 5 we get:

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}}_t \mathbf{x}_0, (1 - \bar{\alpha}_t)\boldsymbol{I}), \text{ where } \bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i. \tag{6}$$

This derivation can be modified to also yield the Gaussian parameterization describing $q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)$. After tedious numerical combinations to combine the three Gaussian terms in Equation 5, we obtain:

$$\begin{aligned} q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\boldsymbol{\Sigma}}_t), \text{ where:} \\ \tilde{\boldsymbol{\mu}}_t &= \frac{\sqrt{\alpha}_t(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)}{1 - \bar{\alpha}_t}\mathbf{x}_0, \\ \tilde{\boldsymbol{\Sigma}}_t &= \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}(1 - \alpha_t)\boldsymbol{I} = \sigma_t^2 \boldsymbol{I}. \end{aligned} \tag{7}$$

We have therefore shown that at each step, $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is normally distributed, with mean $\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0)$ that is a function of $\mathbf{x}_t$ and $\mathbf{x}_0$, and variance $\tilde{\boldsymbol{\Sigma}}_t$ as a function of $\bar{\alpha}_t$ coefficients. In order to match approximately the denoising transition step $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ to ground-truth denoising transition step $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ as closely as possible, we can also model it as a Gaussian. Furthermore, since $\tilde{\boldsymbol{\Sigma}}_t$ is a priori known during training, we can immediately construct the variance of the approximate denoising transition step to also be $\tilde{\boldsymbol{\Sigma}}_t$. Therefore, to define $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$, we only need to find its mean $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$. $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$ is implemented by a neural network that only takes $\mathbf{x}_t$ as an input, and not $\mathbf{x}_0$. This is because $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is conditioned only on $\mathbf{x}_t$ and not $\mathbf{x}_0$.

Under these assumptions, one can show that minimizing the KL divergence in Equation 4 boils down to learning a neural network to predict the original ground truth image $\mathbf{x}_0$ from an arbitrarily noisified version of it $\mathbf{x}_t$.

Going one step further, one can show that this is equivalent to training a neural network $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ that learns to predict the source noise $\boldsymbol{\epsilon}$ that determines $\mathbf{x}_t$ from $\mathbf{x}_0$. This can be understood by rearranging the terms in Equation 2:

$$\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}\right). \tag{8}$$

Reverse Processs Model: The reverse process model $\boldsymbol{\epsilon}_\theta$ is defined in `unet.py` and is an implementation of a CNN called U-Net, as illustrated in Fig. 2. U-Net's role here is to model the denoising function at each step of the reverse diffusion process. The architecture's ability to handle details at multiple scales and its effectiveness in capturing both local and global features make it well-suited for the task of denoising in diffusion models. By predicting the noise that was added at each step of the forward diffusion process, the U-Net helps to gradually reconstruct the data sample from noise.

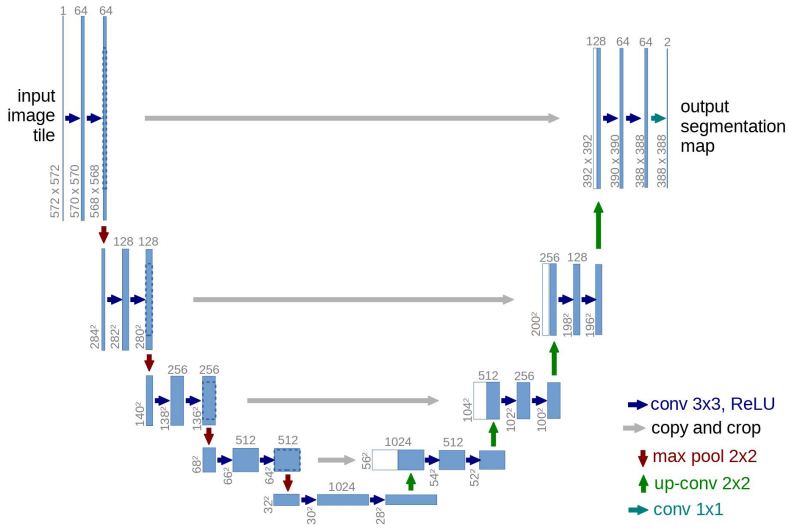

Figure 2: The structure of U-Net.

Training: The training algorithm is described in Alg.1. We utilize a minibatch of data to train our reverse process model, denoted as ϵ_θ, which estimates the noise introduced during the forward diffusion process. You are required to implement the function `forward`, `q_sample` and `p_loss` within the `Diffusion` class. The `p_loss` function defines the training loss using the L_1 loss. Additionally, we set a noise scheduler and pre-define some coefficients in the `__init__` function for efficient reuse, so you should also fill in the blanks there.

Algorithm 1 Training

repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  $t \sim \text{Uniform}(\{1, ..., T\})$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
  $\mathbf{x}_t \leftarrow \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ forward diffusion process
  Take optimizer step on L_1 loss, $\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|_1$
until converged

Sampling: The sampling algorithm is described in Alg. 2. The real implementation considers a minibatch of samples, and use `extract` function to extract coefficients for batched operation. You need to implement function `sample`, `p_sample`, and `p_sample_loop` in the `Diffusion` class that defines the reverse diffusion process to generate images.

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(0, I)$
for $t = T, ..., 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ if $t > 1$, else $\mathbf{z} = 0$
  $\boldsymbol{\epsilon}_t \leftarrow \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ ▷ predicted noise
  $\hat{\mathbf{x}}_0 \leftarrow \frac{1}{\sqrt{\bar{\alpha}_t}} \left(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t\right)$ ▷ estimated $\hat{\mathbf{x}}_0$
  $\hat{\mathbf{x}}_0 \leftarrow \textit{clamp}(\hat{\mathbf{x}}_0, -1, 1)$ ▷ rectify $\hat{\mathbf{x}}_0$
  $\tilde{\boldsymbol{\mu}}_t \leftarrow \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0$ ▷ posterior mean of x_{t-1}
  $\sigma_t^2 \leftarrow \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}(1-\alpha_t)$ ▷ posterior variance of x_{t-1}
  $\mathbf{x}_{t-1} \leftarrow \tilde{\boldsymbol{\mu}}_t + \sigma_t \mathbf{z}$ ▷ reverse diffusion process
return $\mathbf{x}_0$

Evaluation: To gauge the improvement in generative prowess throughout the training process, calculate the Fréchet Inception Distance (FID) between the training dataset and the generated samples from the current model. FID serves as a crucial metric in assessing the quality of generated data, providing a quantitative measure that goes beyond traditional visual inspection.

FID is a widely adopted metric in the realm of generative models, offering a robust evaluation of the dissimilarity between the true data distribution and the generated distribution. By incorporating both the mean and covariance of feature representations extracted from a pre-trained neural network, FID captures nuanced differences and similarities, offering valuable insights into the fidelity of generated samples.

We use `clean-fid` package to easily compute the FID score between resized training images and generated images.

Diffusion Empirical Questions

Clarification: The code to generate the following figures are **already provided**, you can get figures in wandb once you complete the diffusion part.

Part 1: (10 pts)

Training: Plot the training loss of your Diffusion model above over 1,000 training steps with the recommended parameters in the above command line. Your model should be generating blurry cats at this point, similar to the image below. Recommended parameters: `train_steps =1000`, `save_and_sample_every=100`, `fid=False`.

[Expected runtime on Colab T4: 5-10 minutes]

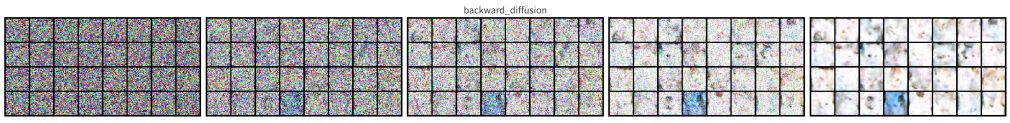

Figure 3: The Backward Diffusion Process, after just 1000 steps.

Part 2: (20 pts)

Training: During training time, the starter code uses the 'compute_fid' function from 'clean-fid' to compute the FID value between training samples and generated samples. Get the FID value every 100 training steps and plot it over 1,000 training steps with the recommended parameters in the above command line. Recommended parameters: `train_steps=1000`, `save_and_sample_every=100`, `fid=True`.

[Expected runtime on Colab T4: 15-60 minutes]

2.1 Part 3: (10 pts)

Visualisation: Visualization: Use the trained model after 1000 steps to illustrate the forward diffusion process on the initial batch of the training dataset at key time intervals: 0%, 25%, 50%, 75%, and 99% of the total timesteps. The resulting figure should resemble the provided sample, though the images will vary due to inherent randomness.

2.2 Part 4: (10 pts)

Visualisation: Use the trained model after 1000 steps to visualize the backward diffusion process. Input the noise images generated from the preceding forward process (i.e., the image from the last timestep in the forward process) to the diffusion model. Utilize these images to generate visualizations of the backward diffusion process at key intervals: 0%, 25%, 50%, 75%, and 99% of the total timesteps. The resulting figure should resemble the provided sample, though the images will vary due to inherent randomness.

Hint: you can find this figure on Colab after calling `visualize_diffusion`

Part 5: (Excerise only, ungraded)

Visualization: Train the model for a full 10,000 iterations and show the images generated in the last sample batch. The images should be a substantial improvement over training with fewer iterations. Recommended parameters: `train_steps=10000`, `save_and_sample_every=1000`, `fid=False`

[Expected runtime on Colab T4: 2 hours]

Part 6: (Excerise only, ungraded)

Visualization: Use the trained model after 10,000 steps to repeat Parts 3 and 4.

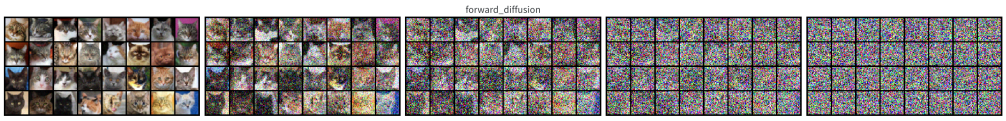

Figure 4: Sample Figure of the Forward Diffusion Process after 10,000 steps.

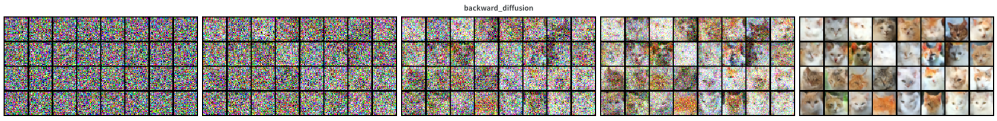

Figure 5: Sample Figure of the Backward Diffusion Process after 10,000 steps..

For this homework, you should upload all the code files that contain your new and/or changed code. Files of type `.py` and `.ipynb` are both fine.