Homework 5-PartA: Generative Adversarial Network

EECS 498-014 Graphics and Generative Models

2024 Fall

Due date: 11:59 P.M., Dec 6, 2024. No late day this time.

Contents

Figure 1: Sample Generated Results that you can get in this homework.

1 Objectives

This assignment is a generative model project on generating realistic images (like dog and cat) by implementing and training one of the most fundamental generative models, the Generative Adversarial Network (GAN). The learning objectives of this assignment are to:

- Gain hands-on experience in GAN architecture and loss function design.
- Learning methods to stabilize and accelerate GAN training by regularization and augmentation techniques.

After completing the tasks, you will be able to generate images like Fig. 1.

Advice at the beginning: **START EARLY!** Even for correct implementation, we expect you to train 3 models, each for 2 hours.

2 Instructions

2.1 Downloading the Code

You can download the starter code here

2.2 Programming Language and Deep Learning Library

In this homework, we will continuously use Python and Pytorch as previous homework. You can choose to complete this homework either on the remote GreatLakes (GL) Server or in your local environment. We will use GreatLakes Server as the standard environment.

2.3 Get access to GreatLakes

GreatLakes (GL) is a high-performance computing cluster in UMich, which provides powerful computation resources. Please follow these steps to connect to GreatLakes.

1. Before connecting to the GreatLakes Server, make sure you are using the campus network or have connected to the network through UMVPN.

2. From your local terminal, type `'ssh ${UNIQUE_NAME}@greatlakes.arc-ts.umich.edu'` to connect to login node of GreatLakes, where `'${UNIQUE_NAME}'` is your UMICH unique name. It requires your UMich account password and Duo Certification via Duo Push or passcodes. More details can be found here. When you log in to VSCode by ssh, you can open the folder to: `/scratch/eecs498s014f24_class_root/eecs498s014f24_class/Your_UniqName`.

3. GreatLakes uses SLURM to manage the computation resource.

   To query compute nodes with GPUs, in the login node, you can execute the following command:

   ```
   salloc --cpus-per-task=6 --gpus=1 --mem-per-gpu=44GB --partition=spgpu --time=0-02:00:00
   --account=eecs498s014f24_class
   ```

   We know that it is hard to copy from the PDF, so we **provide a copy of the execution codes in the README.md**. This command will allocate a **A40 GPU with 42 GB memory for 2 hours**. All the program execution inside this terminal window will be accessible to the GPU (not for other terminal windows). If you close the window of the terminal, the GPU will be released and needs to be requested again. If there are multiple students in this course request at the same time, other students may need to wait. The **maximum number of GPUs that can be allocated by all the students in this course is 10**. Thus, please **start the homework earlier**. The account is shared among all students and the quota for each individual is around **300 hours until the end of the semester**. The storage allocated for each student is around **80GB** (including the conda environment space).

   Note that the terminal window with GPU allocated may be slow in executing the git and conda env commands. We recommend opening another terminal window for these CPU executions.

   Please note that GreatLakes is a shared computation cluster and there are many others using it. Please be considerate to others, including but not limited to:

   - Release the GPU after you are done with your work.
   - Don't run computationally heavy jobs in the login node, which will lead to the login node being slow.

Also, **don't use these GPU resources for personal projects. It will be zero if get caught.**

2.4 Conda Environment Setup

In this homework, and the following homework, we will use conda environment as the Python environment management system. To build a system that is aligned with the testing environment, we need to:

1. Init the conda environment

   ```
   module load python3.9-anaconda
   ```

2. Create a conda environment for this homework by:

   ```
   conda create -n GAN python=3.9

   ```

   Here, the name of this environment will be GAN, and we install Python 3.10 as our interpreter.

3. Activate your environment by:

   ```
   conda activate GAN

   ```

   Then, you will find that the environment signal on the terminal interface is changed from **base** to **GAN**.

4. For PyTorch install, we need to input the following command under the conda environment of **GAN**:

   ```
   pip install torch==2.1.1 torchvision==0.16.1 torchaudio==2.1.1 --index-url https://
   download.pytorch.org/whl/cu118

   ```

   More details can be found here. We don't recommend installing the most up-to-date version of Pytorch. Instead, installing a relatively older version is more stable.

5. Other packages can be installed by the requirements.txt file in the folder by:

   ```
   cd /PATH/TO/YOUR/GAN/HOMEWORK
   pip install -r requirements.txt

   ```

3 Overview

In this homework, you'll work through three tasks to implement the GAN loss function, R1 regularization, and Differential Augmentation. Three papers that are related to the concept are StyleGAN for the architecture, Which Training Methods for GANs do actually Converge? for R1 regularization, and Differentiable Augmentation.

3.1 How to Run the Code?

Before executing the code (training/testing), it is needed to finish all the **TODO**, such that you can run the following line:

```
python train.py --outdir training-runs --data many_shot_dog --use_r1_regularization XXX --use_diffaug XXX  --checkpoint_path XXX
```

Here,

`--outdir` flag refers to the directory where you want to store the training results, including checkpoints, and sample images.

`--data` flag refers to the dataset path.

`--use_r1_regularization` and `--use_diffaug` refers to whether you want to introduce R1 regularization and DiffAug in the training process. It is preset to require users to input these two flags just in case users forget to set the flag. Details about how to set this flag will be provided later.

`--checkpoint_path` refers to the path of checkpoints to load. We provide a pre-trained middle checkpoint (10K iterations) for faster training. Training a full GAN on a small dataset is also very time-consuming (10 hours for decent results on an A40 GPU). Considering the GPU resources available, we only expect to train for two hours (5K iterations) starting from the checkpoint provided.

The generated results and all the logs-related information can be found in folder `training-runs`. A final generation image can be found in Fig. 1.

3.2 Dataset Preparation

The dataset we will use is a few-shot animal dataset, which is roughly 300 images. It can be downloaded by the following command:

```
python dataset_download.py
```

You can also download the dataset manually from Here.

3.3 Checkpoint Preparation

We provide intermediate checkpoints for faster training. You can download it by:

```
wget https://github.com/um-graphics/um-graphics.github.io/releases/download/HW5/checkpoint.zip
unzip checkpoint.zip
```

You can also download it manually from Here.

4 Basic GAN Training

In this part, we will start with the StyleGAN model as our base architecture model.

4.1 Brief Walkthrough to GAN

GAN takes a latent code of $z \in Z$ into the generator G and outputs a generated image $G(z)$. Here, we consider a random noise as our latent code z. Then, the generated images will be discriminated by another neural network, known as Discriminator D, to distinguish if the generated results $G(z)$ are true or false, which is represented as $D(G(z))$. Meanwhile, real samples x will also be input to the Discriminator as $D(x)$ to teach the Discriminator to learn to distinguish between real and fake sample input. Eventually, the goal of the GAN model is to enable the Generator to generate images that can fool the Discriminator that has a strong ability to distinguish real and fake input samples.

4.2 Loss Function

The Loss Function for GAN model is considered two-fold: Generator and Discriminator loss. The loss function of the Generator in StyleGAN can be represented as follows:

$$\mathcal{L}_{\mathcal{G}} = \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1+\exp(-D(G(z))))]. \tag{1}$$

And the loss function for the Discriminator in our GAN is as follows:

$$\mathcal{L}_{\mathcal{D}} = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log(1+\exp(-D(x))] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1+\exp(D(G(z))))]. \tag{2}$$

Task 1: Your first task in this problem is to implement the loss function of the Generator and Discriminator in **TODO `#1, #2, #3,` and `#4`** of the training/loss.py file. Note that it is not needed to care about the regularization for the generator (Perceptual Path Length Regularization) here, which is already implemented. Also, we have handled the Expectation term, which means that the only thing needed to care about is the $\log$ term inside.

Hint: $\log(1+\exp(u))$ is also known as **softplus**.

Execute the code for this part by:

```
python train.py --outdir training-runs --data many_shot_dog --use_r1_regularization False --use_diffaug False --checkpoint_path checkpoint/part1-10K.pkl
```

It is expected to train for 2 hours (about 4.5K iters) which is the maximum time that the GL supports, but feel free to train longer. Each run will create a new subfolder under the folder `training-runs`. Check the one with the biggest ID

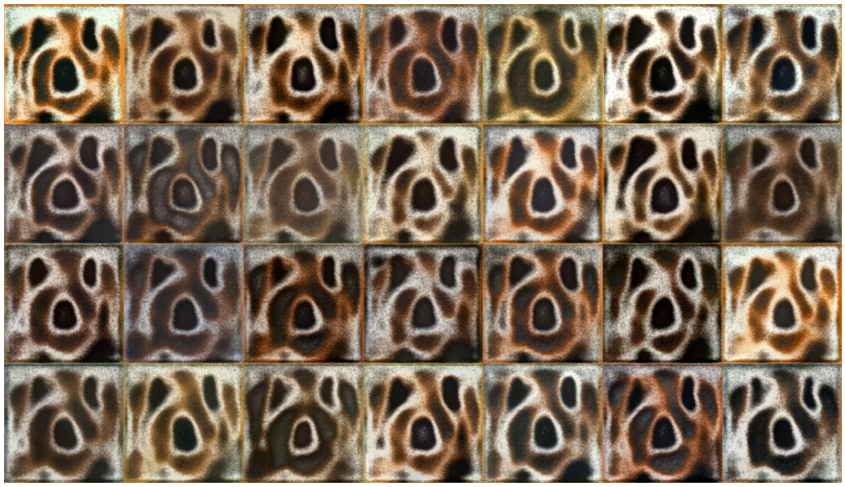

Figure 2: Generated Results without any regularization.

for the current run. If you click `log.txt` file, the first line will show all the flags you used. Submit the generated results (latest `fakesXXXXiter_trained.png` under folder `training-runs`) and your `log.txt` as submission material. The result should have slightly better results than the checkpoint given if implemented correctly. We don't expect too much improvement in the setting without any regularization. Submit the generated results (only one image) and your `log.txt` as submission material.

Task 2: Please explain what is the difference between $\mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}$ term in Eq. 2 and Eq. 1? And why there is such a difference between the Generator and the Discriminator? Briefly explain the difference between the objective of the Generator and the Discriminator loss function is adequate. Write your response in a PDF or TXT file and submit it.

5 R1 Regularization

After training the GAN model in the previous part (if implemented correctly) for 2 hours, you may get visual results similar to Fig. 2.

You may feel that the convergence of the GAN model is so terrible that after hours and hours of training on such a small dataset, we cannot see any signal of the generated images to look like the identity we want.

Then, it is time to introduce regularization terms to the loss function to promote a faster convergence. Regular GAN training without regularization does not always converge to the Nash-Equilibrium, where no agent (Generator and Discriminator here) can improve its expected payoff by deviating from a different strategy. Instead, the introduction of the regularization technique is helpful to change this dilemma.

The first Regularization technique we introduce is the R1 regularization, which is a regularization technique and gradient penalty for training the GAN model. It penalizes the **Discriminator** (Note: no Generator) via penalizing the

gradient on **real data alone**: when the Generator distribution produces the true data distribution and the Discriminator is equal to 0 on the data manifold, the gradient penalty ensures that the Discriminator cannot create a non-zero gradient orthogonal to the data manifold without suffering a loss in the GAN training. Feel free to read more details in section 4.1 of here.

The R1 regularization formula is presented as follows:

$$R_1(\psi) := \frac{\gamma}{2} E_{p_{\mathcal{D}}(x)} \left[\|\nabla D_\psi(x)\|^2 \right], \tag{3}$$

where ψ is the parameter for the Discriminator.

Task 3: Your task is to implement **TODO #5** of the training/loss.py file. The gradient term of Discriminator, $\nabla D_\psi(x)$, is provided.

When you feel that the code is implemented correctly, you can set flag `--use_r1_regularization True` to bring in the R1 regularization in the training. You should see faster convergence within the same number of training iterations.

Execute the code for this part by:

```
python train.py --outdir training-runs --data many_shot_dog --use_r1_regularization True --use_diffaug False --checkpoint_path checkpoint/part2-10K.pkl
```

It is expected to train for 2 hours (about 4.5K iters) which is the maximum time that the GL supports, but feel free to train longer. Each run will create a new subfolder under the folder `training-runs`. Check the one with the biggest ID for the current run. If you click `log.txt` file, the first line will show all the flags you used. Submit the generated results (latest `fakesXXXXiter_trained.png`) and your `log.txt` as submission material. The result should have slightly better or similar results than the checkpoint given if implemented correctly. The grade is more dependent on whether you have a correct implementation and stable performance. Submit the generated results (only one image) and your `log.txt` as submission material.

Task (optional extra credits): Despite the two regularization techniques we have shown above, what are some other augmentation or regularization techniques that are functional in GAN? Please search for the literature online and list three or more here.

6 Differentiable Augmentation

By successfully training the previous section, you may generate a result similar to Fig. 3. Though the generated results look much better than the version without regularization, it is still hard to reach similar visual results as Fig. 1 with the same amount of training complexity. Next, we will introduce helpful augmentation tricks in the GAN from Differ-

Figure 3: Generated Results with R1 regularization.

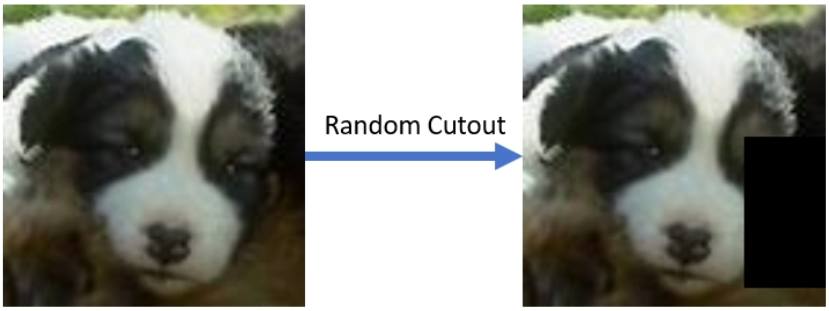

Figure 4: A sample result for the Random Cutout.

entiable Augmentation. Data augmentation is a common trick in machine learning that increases the diversity of the limited training dataset. Some popular methods for images include rotation, flipping, resizing, etc.

Task 4: Implement the random cutout function (`rand_cutout`) in the `DiffAugment_pytorch.py` file. You can check your augmented images directly by executing:

```
python DiffAugment_pytorch.py
```

If you implement it correctly, Fig. 4 will be a possible random cutout result. The mask bounding box range is recommended to be randomly chosen in 25%-50% of the height and the width, and the top left corner position is random. Despite having a correct store image while running the `DiffAugment_pytorch.py`, you also need to ensure that the training with multiple batch sizes is successful.

Task 5: The second task is to implement the idea behind the Differentiable Augmentation (DiffAug). DiffAug shows that applying data augmentation on the following three places at the same time exerts much better visual results within the same amount of training (see Fig 5): on the fake samples generated by the generator, and on both real and fake (generated) samples for the Discriminator. However, it is unnecessary to implement three parts. Instead, one implementation on Discriminator (check `run_D` function of `training\loss.py`) is enough. This task only requires

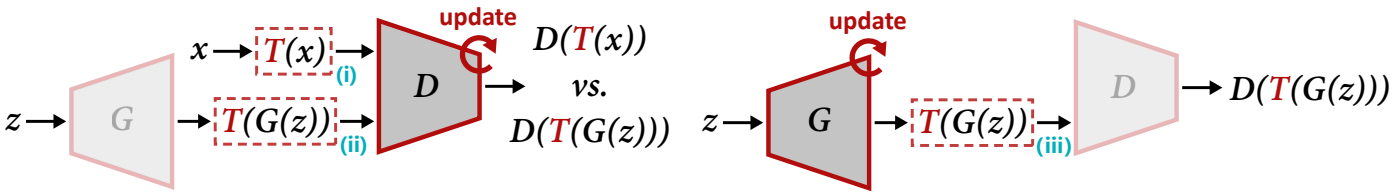

Figure 5: **Overview of DiffAug** for updating Discriminator D (left) and Generator G (right). DiffAug applies the augmentation function T to both the real samples x and the generated output $G(z)$. When we update G, gradients need to be back-propagated through T, which requires T to be differentiable w.r.t. the input.

you to execute the following code to train the model starting from the checkpoint provided and show the generated results.

Execute the code for this part by:

```
python train.py --outdir training-runs --data many_shot_dog --use_r1_regularization True --use_diffaug True --checkpoint_path checkpoint/part3-10K.pkl
```

It is expected to train for 2 hours (about 3K iters) which is the maximum time that the GL supports, but feel free to train longer. Each run will create a new subfolder under the folder `training-runs`. Check the one with the biggest ID for the current run. If you click `log.txt` file, the first line will show all the flags you used. Submit the generated results (latest `fakesXXXXiter_trained.png`) and your `log.txt` as submission material. The result should have slightly better or similar results than the checkpoint given if implemented correctly. The grade is more dependent on whether you have a correct implementation and stable performance. Submit the generated results (only one image) and your `log.txt` as submission material.

Task (optional extra credit): Checkout lines 56-59 of `training/loss.py`. Think about why we only need to modify one part and it is enough to handle three cases shown in Fig. 5. Share your thoughts here.

7 Submission and Grading

HW5 has two parts: GAN and Diffusion which occupy 40% and 60% points respectively. The optional task is worth 10 % points in total (also due on the same date). For submission, please combine the submission materials from two parts into one single .zip file. All 5 compulsory tasks are worth points equally.

To submit your assignment, please zip **your code** as well as the **generated images** (based on the task requirement) and the **required free response question** to Canvas. We will read your implementation in the snippet range given and **a correct implementation is equivalently important as an improved visual result**. Please don't include any dataset images, or trained checkpoints in your uploaded zip file. The file size should be less than 10 MB.

Homework 5-PartB: Diffusion Model

EECS 498-014 Graphics and Generative Models

2024 Fall

Due date: 11:59 P.M., Dec 6, 2024. No late day this time.

Contents

1 Objectives

This assignment is a generative model project on generating images by implementing and inference Diffusion Models. The learning objectives of this assignment are to:

- DDPM Denoise Scheduler.
- DDIM Denoise Scheduler.
- DPS Loss for test-time optimization
- SDS and VSD loss for rendering image enhancement.

All codebase of the Diffusion Model project shares one. Later tasks depend on the successful implementation of the former tasks. For example, DPS loss requires a correct implementation of the DDIM denoise scheduler; thus, it is recommended to do the tasks sequentially.

Advice at the beginning: **START EARLY!**

2 Instructions

2.1 Downloading the Code

You can download the starter code here

2.2 Programming Language and Deep Learning Library

In this homework, we will use Python as our sole programming language and the deep learning library will be Pytorch. You can choose to complete this homework either on the remote GreatLakes (GL) Server or in your local environment. We will use GreatLakes Server as the standard environment, but you are encouraged to first try setting up the project in your local environment for a better coding experience.

2.3 Conda Environment Setup

In this homework, and the following homework, we will use conda environment as the Python environment management system. To build a system that is aligned with the testing environment, we need to:

1. Create a conda environment for this homework by:
   ```
   conda create -n Diffusion python=3.10
   ```
   Here, the name of this environment will be Diffusion, and we install Python 3.10 as our interpreter.

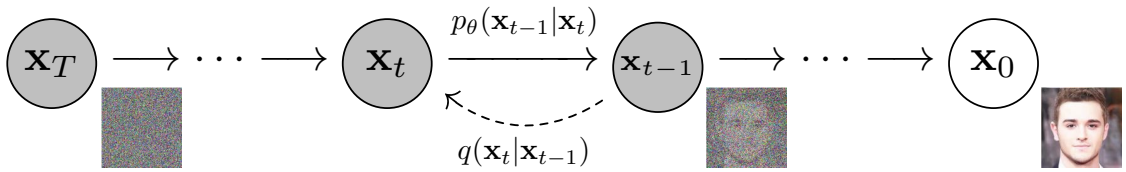

Figure 1: The Diagram of Diffusion Model from DDPM.

2. Activate your environment by:

   `conda activate Diffusion`

   Then, you will find that the environment signal on the terminal interface is changed from **base** to **Diffusion**.

3. For PyTorch installation, we need to input the following command under the conda environment of **Diffusion**:

```
pip install torch==2.1.1 torchvision==0.16.1 torchaudio==2.1.1 --index-url https://download.pytorch.org/whl/cu118

```

   More details can be found here. We don't recommend installing the most up-to-date version of Pytorch. Instead, installing a relatively older version is more stable.

4. Other packages can be installed by the requirements.txt file in the folder by:

```
cd Diffusion_homework/
pip install -r requirements.txt

```

   Note that, please strictly follow the version in `requirements.txt`, **the diffusers version we provide is crucial to execute some code (e.g., VSD) correctly. Don't directly use conda env from GAN.**

3 Diffusion Problem Statement

In this section, we will use pre-trained weights to run the inference of various diffusion models. Since the diffusion model training is generally very slow and we already have adequate training in the GAN section, the following sections only include modifications that require running inference to verify the correctness. The pre-trained weight should be automatically downloaded to the cache of your computer/server. Sec. 4 will require you to implement the DDPM scheduler. Sec. 5 will require you to implement the DDIM scheduler. Sec. 6 will require you to implement DPS loss. Sec. 7 and Sec. 8 will require you to implement SDS and VSD.

3.1 Brief Walkthrough of Diffusion Model

In the forward process (aka diffusion process) of the diffusion model, we add random Gaussian noise to a clean image $\mathbf{x}_0$, and the noisy form is denoted as $\mathbf{x}_t$ which is at timestep t. Since this homework only considers the inference part, it

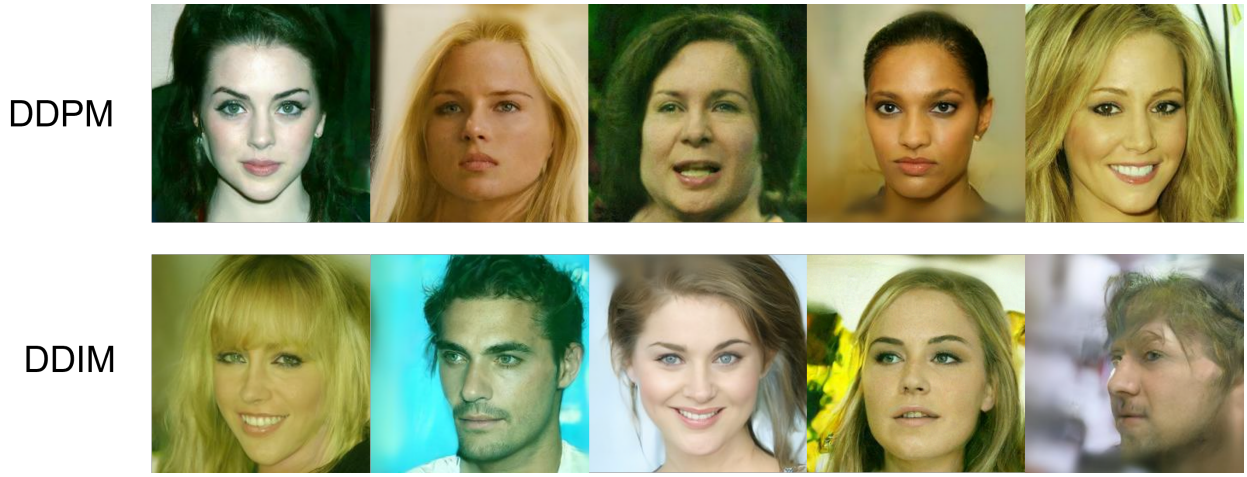

Figure 2: Sample Generated Results on the FFHQ pre-trained Diffusion Model. DDPM uses 1000 steps and DDIM uses 50 steps.

is not necessary to implement the forward process of the diffusion model for the training. In the inference stage, we will directly start with a pure noise image $\mathbf{x}_T$, as shown in Fig. 1 (from right to left perspective).

In the reverse process (aka denoising process), the target is to obtain $\mathbf{x}_{t-1}$ from $\mathbf{x}_t$ until reaching $\mathbf{x}_0$. In this process, the neural network is employed to estimate noise that was added to the image at timestep t. In DDPM and DDIM, we will use the output of the UNet as the source to calculate $\mathbf{x}_0$, as shown in Fig. 1 (from left to right perspective). Specific algorithms and details are provided in the sections below.

4 DDPM Denoise

Though there may be lots of denoise schedulers in the public AIGC software, one of the most famous and widely used denoise must be DDPM. In this section, we will first walk through specific algorithm inside DDPM and then you will need to fill in **TODO** note in `scheduler/DDPM_scheduler.py`

In this homework, we utilize epsilon prediction from DDPM to estimate clean image $\mathbf{x}_0$, which can be represented by:

$$\mathbf{x}_0 \approx \hat{\mathbf{x}}_0 = \left(\mathbf{x}_t - \sqrt{\bar{\beta}_t}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t)\right) / \sqrt{\bar{\alpha}_t}, \tag{1}$$

where $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t)$ is the model output of neural network. This formula means that we estimate a clean image $\mathbf{x}_0$ by using a network to predict the noise, based on the current sample $\mathbf{x}_t$ at the timestep t.

With estimated clean image $\mathbf{x}_0$, we can use it along with $\mathbf{x}_t$ to predict previous sample $\mathbf{x}_{t-1}$. The formula can be

expressed below:

$$\mathbf{x}_{t-1} := \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{\bar{\beta}_t}\mathbf{x}_0 + \frac{\sqrt{\alpha_t}(\bar{\beta}_{t-1})}{\bar{\beta}_t}\mathbf{x}_t + \tilde{\sigma}_i z, \tag{2}$$

where $z \sim N(\mathbf{0}, \boldsymbol{I})$ is a random noise and $\tilde{\sigma}_i z$ is a small disturbance noise (already implemented). Be careful that, we are doing the inference of diffusion model here, so the timestep should be from the maximum time step (e.g. 1000) to 0, which means that $\mathbf{x}_t$ is the previous calculation result than the current $\mathbf{x}_{t-1}$.

To make the life easier, we provide the calculation for each variable below:

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{3}$$

$$\bar{\alpha}_{t-1} = \prod_{s=1}^{t-1} \alpha_s \tag{4}$$

$$\bar{\beta}_t = 1 - \bar{\alpha}_t \tag{5}$$

$$\bar{\beta}_{t-1} = 1 - \bar{\alpha}_{t-1} \tag{6}$$

$$\alpha_t = \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \tag{7}$$

$$\beta_t = 1 - \alpha_t \tag{8}$$

To calculate $\bar{\alpha}_t$ and $\bar{\alpha}_{t-1}$, you need to look at lines 220-222 to know how the cumulative product from the first to the last is collected. Then, think about how to use the provided variables `t` and `prev_t` to access current and previous timestep. By default, the DDPM sampling steps are 1000 and there is no need to set that manually. The task that the model is doing is to unconditionally generate a human face. This is a pre-trained weight that has trained a large human face dataset and it can generate a random face without giving any conditional input (like language prompt).

Action: Fill in **TODO** `#1`, `#2`, `#3` in `scheduler/DDPM_scheduler.py`

To execute the code, you can run the following command:

```
python execute_DDPM_DDIM.py --scheduler_name DDPM
```

Then, you can find some generated results in the `DDPM_results` folder. Some sample-generated results can be found in Fig. 2. We only expect the generated images to look like human faces (low-level details, like background color, are minor concerns), and then we could give full points (also a correct implementation of the algorithm). You can also generate more than 10 samples and select 10 samples that you feel good to submit.

Task1: Implement Equations correctly and submit 10 of your generated images.

5 DDIM Denoise

While you run the Diffusion Model with the DDPM scheduler, you may feel that the wait time is too long. And the next question that readers may ask is how to increase the speed of the inference. Certainly, better GPU and parallel computation can be helpful. However, algorithm-wise, decreasing the number of denoise steps is one of the most popular ways that researchers are working on. In this section, we will continue to introduce a speed-up version of the DDPM, which is DDIM (Denoising Diffusion Implicit Models). The denoising steps will be decrease from 1000 to 50 steps.

To apply DDIM in the inference, it is not needed to re-train a new diffusion model with DDIM as the scheduler. Instead, we can keep the DDPM-trained weights and switch to DDIM in the inference stage only. Thus, for this section, we keep using the same pre-trained weight.

The formula for the $\boldsymbol{x}_{t-1}$ calculation can be represented by:

$$\boldsymbol{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \underbrace{\left(\frac{\boldsymbol{x}_t - \sqrt{\bar{\beta}_t}\epsilon_\theta(\boldsymbol{x}_t)}{\sqrt{\bar{\alpha}_t}} \right)}_{\text{“ predicted } \boldsymbol{x}_0\text{”}} + \underbrace{\sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} \cdot \epsilon_\theta(\boldsymbol{x}_t)}_{\text{“direction pointing to } \boldsymbol{x}_t\text{”}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}} \tag{9}$$

where:

$$\bar{\beta}_t = 1 - \bar{\alpha}_t \tag{10}$$

Here, σ_t is already provided, but the rest requires implementation in the codebase. The predicted x_0 is the same as that in DDPM. Random Noise term $\sigma_t \epsilon_t$ is already provided. By default, the DDIM sampling steps are 50 and there is no need to set that manually.

Action: Fill in **TODO** `#1` and `#2` in `scheduler/DDIM_scheduler.py`.

The execution codebase of DDPM and DDIM is shared. To execute the code, you can run the following line:

```
python execute_DDPM_DDIM.py --scheduler_name DDIM
```

Then, you can find some generated results in the `DDIM_results` folder. Some sample-generated results can be found in Fig. 2. We only expect the generated images to look like human faces (low-level details, like background color, are minor concerns), and then we could give full points (also a correct implementation of the algorithm). We know that the pre-trained weight is not a strong model, so you can generate more than 10 samples and select 10 samples that you feel good about submission.

Task2: Please implement the equations and provide 10 of your generated images.

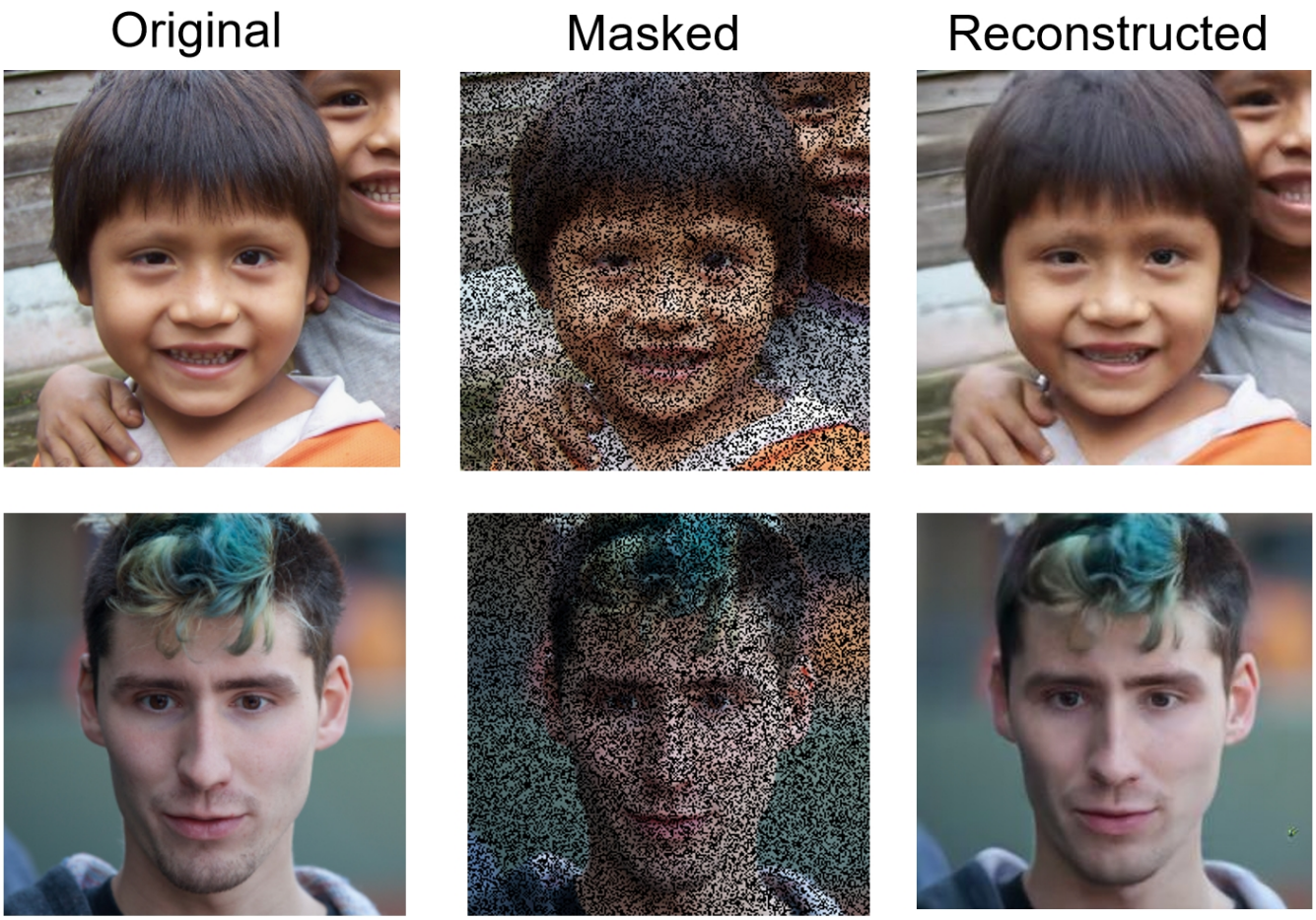

Figure 3: Sample generated results of DPS loss in inpainting task.

Task3: Please report the time you need for DDPM and DDIM respectively and briefly talk about the pros and cons of using DDIM as the scheduler over DDPM. One sentence response is adequate.

6 DPS Loss

In this section, you will implement a test-time optimization algorithm for the inverse problem. Inverse problems in computer vision target reverting the underlying causes or inputs from observed data. The formation of the inverse problem can be broadly defined as solving: $\mathbf{y} = \mathcal{A}(\mathbf{x}) + \mathbf{n}$ where $\mathbf{x}$ is the observed data and $\mathbf{n}$ can be noise or error term. The target is to solve $\mathbf{x} = \mathcal{A}^{-1}(\mathbf{y})$. Famous task of inverse problems includes Super-Resolution (generating a high-resolution image from a low-resolution one), DeBlurring (reverting blurred images to the sharp, clear original form), InPainting (filling in the missing pixel contents of the input images), etc.

Diffusion Posterior Sampling, DPS, is a simple test time optimization method. Test time optimization refers to tricks that are deployed in the testing stage directly without the need for training or fine-tuning. After obtaining $\boldsymbol{x}_{t-1}$ at the end of DDPM and DDIM, DPS only requires extra calculation as the following formula:

$$\boldsymbol{x}_{t-1} = \boldsymbol{x}_{t-1} - \alpha_t \nabla_{\boldsymbol{x}_t} \|\mathbf{y} - A(\mathbf{x}_0)\|_2, \tag{11}$$

where $\mathbf{x}_0$ is the **estimated** original clean sample (check Equation (9) for details) which is the same calculation as you do in DDPM and DDIM. $\mathbf{y}$ is degraded data by $\mathcal{A}(\mathbf{x}) + \mathbf{n}$, which shares the same operator $\mathcal{A}$ with $\mathbf{x}_0$. The formula is a simplified form of what you see from the paper for clarity of implementation purposes. More details can be found in the Appendix D of the original paper. α_t is a constant scaling factor provided. $\nabla_{\boldsymbol{x}_t}$ is the sum of the gradients with respect to $\boldsymbol{x}_t$. We recommend using the `torch.autograd.grad`. Be careful that, we are doing the inference here, so the timestep should be from t to 0, which means that $\mathbf{x}_t$ is the previous calculation result than the current $\mathbf{x}_{t-1}$. The sample can be found in Fig. 3. All gradient-related implementations are provided. Inpainting algorithms are provided by default setting, so don't modify the hyperparameters of it. DPS will share the scheduler with DDIM and you will need to implement it inside the DDIM scheduler.

Fill in **TODO** `#3` in `scheduler/DDIM_scheduler.py`.

To execute the code, you can run the following line:

```
python execute_DPS.py
```

DPS codebase will automatically read 3 images provided in the folder `__asset__`, we only consider these three images for grading. After the execution, you will see a folder named `DPS_results` with all the pair datasets needed.

Task4: Implement the equations correctly and provide all images in the `DPS_results` folder (masked images and the inpainted images).

Task5: **Question**: What is the extra time cost for applying the DPS compared to DDIM? What will happen if we set fewer timesteps than usual (look at **TODO** in `execute_DPS.py`)? Please run the experiment in different timesteps and share your findings. The response can be as short as one sentence. Sticking some figures is recommended.

7 Score Distillation Sampling (SDS) Loss

In this section, we will implement two text-to-3D algorithms (but our focus will be their image perspective actually), which are Score Distillation Sampling (**SDS**) and Variational Score Distillation (**VSD**). Sample visual results can be found in Fig. 4. There may be a light training process for each instance and the peak GPU memory cost can come to 16 GB. Due to the massive time cost of 3D rendering and GPU memory cost, we generate images of SDS and VSD in 2D space by using an identity rendering function that isolates other 3D factors. This process also replicates Section 3.3 and Fig.3 of the **VSD paper**. Note that, the SDS and DPS are all based on Stable Diffusion 2.1 weight, and the pre-trained weight

Figure 4: Sample generated results of SDS and VSD. The text prompt as condition input is "a photograph of an astronaut riding a horse". Try to observe the difference between each guidance scale in each method.

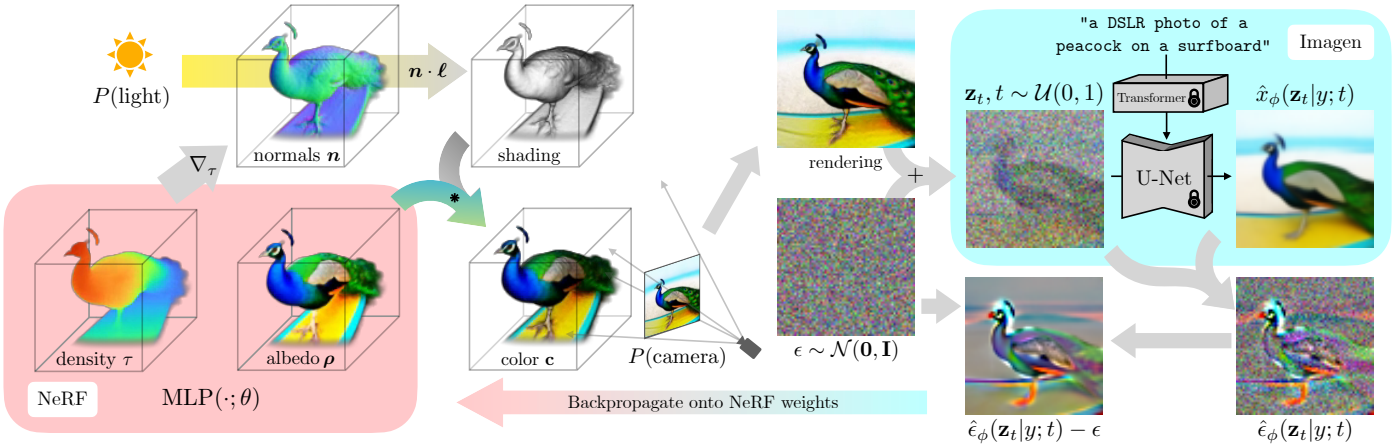

Figure 5: **SDS Pipeline**.

will be downloaded to your cache location automatically the first time you execute the code. Meanwhile, it is needed to be careful with the storage available in GreatLakes (80GB budget in total).

The strength of SDS and VSD is that, without 3D training data, they can still perform text-to-3D synthesis by using a 2D pre-trained diffusion model as a prior for optimization of a parametric image generator.

Traditional SDS. The idea of the SDS origins from the score function perspective of the diffusion model: the gradients of diffusion loss $\nabla_\theta \mathcal{L}_{\text{Diff}}$ could be regarded as an updating direction from the region with (a less realistic image) to a region with more realistic image. Intuitively, if we could optimize each rendered view of 3D using the $\nabla_\theta \mathcal{L}_{\text{Diff}}$ to reach the high-score regions, then hopefully the obtained 3D structures will be realistic as well.

As shown in Fig. 5, the analysis of the SDS Loss should be started from the text-to-image model, which is how U-Net predicts the injected noise $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t)$ at time step t. Here, as we are using Stable Diffusion, z_t is the latent feature for input image $\mathbf{x}_t$, which is the output of the encoder. Although we are doing SDS in the latent space (z), it is also feasible

to use SDS in pixel space when using a diffusion model without latent spaces, like DeepFloyd.

Next, SDS initializes a NeRF-like model with random weights, then repeatedly renders views of that NeRF from random camera positions and angles, which constructs **rendering** images x as the input to the diffusion model. In the diffusion model, the loss function calculates the mean square error difference between the added noise ϵ and the injected noise $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t)$ that is conditioned on timestep t and language y. Subtracting the injected noise produces an update direction $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t) - \epsilon$ that is backpropagated through the rendering process to update the NeRF MLP parameters. The gradient of $\mathcal{L}_{\text{Diff}}$ can be represented as below:

$$\nabla_\theta \mathcal{L}_{\text{Diff}}(\phi, \mathbf{x} = g(\theta)) = \mathbb{E}_{t,\epsilon}\left[w(t) \underbrace{\left(\hat{\epsilon}_\phi(\mathbf{z}_t|y,t) - \epsilon\right)}_{\text{Noise Residual}} \underbrace{\frac{\partial \hat{\epsilon}_\phi(\mathbf{z}_t|y,t)}{\mathbf{z}_t}}_{\text{U-Net Jacobian}} \underbrace{\frac{\partial \mathbf{x}}{\partial \theta}}_{\text{Generator Jacobian}} \right], \tag{12}$$

where $g(\theta)$ means a differentiable generator g transforms parameters θ (in the paper, this is NeRF MLP parameters) to create an image $\mathbf{x} = g(\theta)$. The UNet Jacobian term refers to the Jacobian of the output with respect to the input at the UNet model. Generator Jacobian is the Jacobian with respect to NeRF parameter θ. Details of $w(t)$ will be provided later. A further contribution of the SDS paper is that they empirically found that omitting the U-Net Jacobian term also leads to good results but saves a lot of computation in the UNet backpropagation.

$$\nabla_\theta \mathcal{L}_{\text{SDS}}(\phi, \mathbf{x} = g(\theta)) \triangleq \mathbb{E}_{t,\epsilon}\left[w(t) \left(\hat{\epsilon}_\phi(\mathbf{z}_t|y,t) - \epsilon\right) \frac{\partial \mathbf{x}}{\partial \theta} \right]. \tag{13}$$

Intuitively, the difference between the predicted and added noise $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t) - \epsilon$ is roughly an update direction of $\mathbf{z}_t$ obtained during diffusion process, which is toward to the desired direction (better image and text-condition alignment). As $\mathbf{z}_t$ (with added noise) is the input to the UNet, we can regard $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t) - \epsilon$ as a (pseudo) gradient for $\mathbf{z}_t$, and naturally omit the U-Net Jacobian part as we already obtain the gradients for the input of U-Net.

Generator Jacobian $\frac{\partial \mathbf{x}}{\partial \theta}$ is the gradient term to the NeRF parameter θ. $w(t)$ is a weighting function that depends on the timestep t. There can be various ways to define the $w(t)$, we instead choose the default option from the paper:

$$w(t) = \sigma_t^2 \tag{14}$$

$$\sigma_t = \sqrt{1 - \alpha_t^2} \tag{15}$$

Our 2D version. Due to the slow rendering speed and limited resources that we can provide, we only require a 2D version to implement. In our 2D version, we only optimize a single image $\mathbf{x}$ by using SDS loss to satisfy our generation requirements, instead of NeRF. In this setting, we parameterize $\mathbf{x} = \theta$, which means we directly optimize the value of $\mathbf{x}$ as the image generator parameter θ, and naturally, $\frac{\partial \mathbf{x}}{\partial \theta} = 1$ under this settings.

The overall formula can be simplified to the following in our 2D case:

$$\nabla \mathcal{L}_{\text{SDS}}(\phi, \mathbf{x}) \triangleq \mathbb{E}_{t,\epsilon}\left[w(t)\left(\hat{\epsilon}_\phi(\mathbf{z}_t|y,t) - \epsilon\right)\right], \tag{16}$$

The implementation of the backpropagation update to the gradient term $\nabla \mathcal{L}_{\text{SDS}}$ is provided. This is done by calculating the mean square error (MSE loss) between $\mathbf{x}$ and $\mathbf{x} - \nabla \mathcal{L}_{\text{SDS}}$, which leads to the gradient of loss equal to $\mathbf{x} - (\mathbf{x} - \nabla \mathcal{L}_{\text{SDS}}) = \nabla \mathcal{L}_{\text{SDS}}$. Feel free to look at the comment of lines after TODO #1 for further details.

Action: Fill in **TODO** #1 in `execute_SDS_VSD.py` and **TODO** #1 and #2 in `utils/sds_vsd_utils.py`.

To execute the code, you can run the following command:

```
python execute_SDS_VSD.py --generation_mode=sds --store_dir=SDS_results
```

All other hyperparameters, like learning rate, pre-trained base model, height, width, and guidance scale are fixed for this generation mode. Then, after executing, there will be a folder started with `"SDS_results"`. The file named `generated_image.png` is the final generated results and you will need to submit the `generated_progressive.png` file, which contains progressive information of the generation.

Next, we introduce Classifier Free Guidance (CFG). CFG is a test-time technique for trading off the quality and diversity of the samples, which modifies the model by $\hat{\epsilon}_\phi(\mathbf{z}_t, t, y) := \epsilon_\phi(\mathbf{z}_t, t, \varnothing) + s * \epsilon_\phi(\mathbf{z}_t, t, y)$, where $\varnothing$ is an "empty" string input as a placeholder for the text prompt which represents the unconditional case (like human face generation in DDPM and DDIM task that we have done), and $s > 0$ is a scaler factor as the guidance scale. The core concept is that we want the conditional generation to be far away from the unconditional ones. A larger guidance scale usually improves the text-image alignment but reduces diversity. The default execution setting we have tried used the guidance scale of 7.5, which is the most widely used CFG setting for Text-to-Image tasks. Further, execute the following command, which is a reasonable setting for SDS to have decent visual:

```
python execute_SDS_VSD.py --generation_mode=sds --store_dir=SDS_results --guidance_scale=100
```

The result should look like the second one in Fig. 4.

Task 6: Implement the code correctly and submit two `generated_progressive.png` with the text prompt provided.

8 Variational Score Distillation (VSD) Loss

Based on the previous section, VSD loss is an improved form of the SDS, coming with a rich structure and better visual effects. VSD treats the corresponding 3D scene as a random variable instead of a single point as in SDS. Another advantage of VSD is that VSD can apply regular CFG guidance scale values (7.5) to achieve decent results. Using CFG

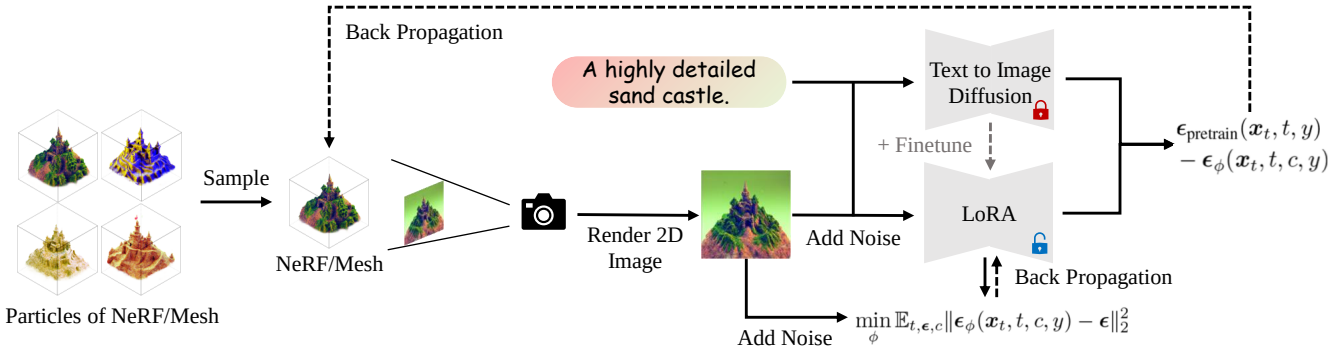

Figure 6: **VSD Pipeline**.

value as big as 100 like SDS generates over-saturating and over-smoothing results as shown in Fig. 4. To model the score of the variational distribution, VSD needs to train an additional diffusion model parameterized by LoRA. LoRA is a widely used fine-tuning strategy in the Diffusion Model and LLM that can quickly adapt a new sample dataset without tuning a lot of parameters. With a small number of iterations for LoRA training, we can reach decent effects. The specific implementation of LoRA is provided. The overall pipeline can be seen from Fig. 6, where the rendered image is sent to the pre-trained diffusion model, and the score of the variational distribution (LoRA) to compute the gradient of VSD.

The corresponding gradient of VSD in our 2D case can be represented as:

$$\nabla \mathcal{L}_{\text{VSD}}(\phi, \mathbf{x}) \triangleq \mathbb{E}_{t,\epsilon}\left[w(t)\left(\hat{\epsilon}_\phi(\mathbf{z}_t; y, t) - \epsilon_\phi(\mathbf{z}_t; y, t)\right)\right]. \tag{17}$$

Note, for the simplicity of explanation and the alignment to the previous formula, we make some modifications to the original formula in the paper. The core concept here is that we are training a LoRA to model the current fake distribution. The main difference we want to strengthen is the change from random noise ϵ to LoRA predicted noise $\epsilon_\phi(\mathbf{x}_t, y, t)$.

Action: Fill in **TODO #3** `utils/sds_vsd_utils.py`. The rest shares with SDS and you need to ensure that the SDS implementation is correct.

To execute the code, you can run the following line:

```
python execute_SDS_VSD.py --generation_mode=vsd --store_dir=VSD_results
```

All other hyperparameters, like learning rate, pre-trained base model, height, width, and guidance scale are fixed for this generation mode. Then, after executing, there will be a folder started with `"VSD_results"` occurs. The file named `generated_image.png` is the final generated results and you will need to submit the `generated_progressive.png` file, which contains progressive information of the generation.

Task7: Implement the equations and submit the `generated_progressive.png` with the default text prompt provided.

9 Submission and Grading

HW5 has two parts: GAN and Diffusion which occupy 40% and 60% points respectively. The optional task is worth 10% points in total (also due on the same date). For submission, please combine the submission materials from two parts into one single .zip file. All 7 compulsory tasks are worth points equally.

To submit your assignment, please zip **your code** as well as the **generated images** (based on the task requirement) and the **required free response question** to Canvas. We will read your implementation in the snippet range given and **a correct implementation is equivalently important as a good visual result**. Please don't include any dataset images, or trained checkpoints in your uploaded zip file. The file size should be less than 10 MB.