XCS236 Problem Set 4

Due Sunday, November 23 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs236-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Instructor Note

You will submit the entire `submission` directory. To do so run the command below from the `src` directory

```
bash collect_submission.sh
```

or you can choose to zip the folder up manually.
Your submission zip should contain:

1. `src/submission/__init__.py`
2. `src/submission/inpaint.py`
3. `src/submission/sample.py`
4. `src/submission/score_matching_utils.py`
5. `src/submission/score_matching.py`

1 Score Matching

Consider a univariate Gaussian distribution $p(x) = \mathcal{N}\left(x \mid \mu, \sigma^2\right)$, where μ is the mean and σ^2 is the variance of the distribution.

(a) **[2 points (Written)] Derive the score function**

The score function is defined as the gradient of the log-density function with respect to x, i.e., $\nabla_x \log p(x)$. Derive the score function for the univariate Gaussian distribution.

(b) **[4 points (Written)] Score Matching Loss**

The score matching loss is defined as:

$$L(\theta) = \mathbb{E}_{p(x)}\left[\frac{1}{2}\left\|\nabla_x \log p(x) - \nabla_x \log q_\theta(x)\right\|^2\right] \tag{1}$$

where $p(x)$ is the true distribution, and $q_\theta(x)$ is a model distribution parameterized by θ.

Using the score function derived in part a, compute the score matching loss for the univariate Gaussian model, assuming that $p(x) = \mathcal{N}(x \mid \mu, \sigma^2)$ and $q_\theta(x) = \mathcal{N}(x \mid \mu_\theta, \sigma_\theta^2)$. We are looking for a final derivation $A^2(\mu^2 + \sigma^2) + 2AB\mu + B^2$ where A and B are expressed in terms of σ, μ, μ_θ, and σ_θ.

(c) **[2 points (Written, Extra Credit)] Generalization to Multivariate Gaussian**

Extend the result of part b. to a multivariate Gaussian distribution $p(\mathbf{x}) = \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$. Derive the expression for the score matching loss for the multivariate case, and explain the challenges posed by the covariance matrix $\boldsymbol{\Sigma}$ in higher dimensions.

2 Concrete Score Matching

Concrete Score matching is a framework for estimating discrete probability distributions by learning a score function tailored to discrete data. It extends traditional score matching, which is originally designed for continuous data, to handle discrete variables by leveraging neighborhood structures.

The **neighborhood mapping** $\mathcal{N}$ identifies all data points that are considered neighbors of a particular example $\mathbf{x}$. Formally, we denote $\mathcal{N}(\mathbf{x} : \mathcal{X} \to \mathcal{X}^K$ as the function mapping each example $\mathbf{x} \in \mathcal{X}$ to a set of neighbors, such that $\mathcal{N}(\mathbf{x}) = \{\mathbf{x}_{n_1}, ..., \mathbf{x}_{n_k}\}$ and $K =\mid \mathcal{N}(\mathbf{x}) \mid$.

In this problem, we will consider a discrete data setting of length d: $\mathcal{X} = \{0, 1\}^d$. For each $\mathbf{x} \in \mathcal{X}$, the neighborhood $\mathcal{N}(\mathbf{x})$ consists of all vectors that differ from $\mathbf{x}$ by exactly one bit.

More formally, In other words, two vectors are neighbors if their **Hamming distance** is equal to 1.

$$\mathcal{N}(\mathbf{x}) = \{\mathbf{x}' \in \mathcal{X} : \text{Hamming}(\mathbf{x}, \mathbf{x}') = 1\} \tag{2}$$

Then, the **Concrete score** $c_{p_{\text{data}}}(\mathbf{x}; \mathcal{N}) : \mathcal{X} \to \mathbb{R}^{|\mathcal{N}(\mathbf{x})|}$ for a given distribution $p_{\text{data}}(\mathbf{x})$ evaluated at $\mathbf{x}$ is:

$$c_{p_{\text{data}}}(\mathbf{x}; \mathcal{N}) \triangleq \left[\frac{p_{\text{data}}(\mathbf{x}_{n_1}) - p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x})}, ..., \frac{p_{\text{data}}(\mathbf{x}_{n_k}) - p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x})} \right]^\top \tag{3}$$

The objective of Concrete Score Matching is to learn a score function that minimizes the discrepancy between the estimated score and the true score derived from the data distribution. This is achieved by defining an objective function that quantifies this discrepancy:

$$\mathcal{L}_{\text{CSM}}(\theta) = \sum_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \left\| c_\theta(\mathbf{x}; \mathcal{N}) - c_{p_{\text{data}}}(\mathbf{x}; \mathcal{N}) \right\|_2^2 \tag{4}$$

where $c_\theta(\mathbf{x}; \mathcal{N})$ is the parameterized score function we aim to learn.

(a) **[4 points (Written)] Learning Objective**

Simplify the learning objective for concrete score matching with a Hamming distance of 1 to remove dependency on the concrete score of the true data distribution $c_{p_{\text{data}}}$.

(b) **[4 points (Written)] Tractability**

In our discrete space where $\mathcal{X} = \{0, 1\}^d$, consider the Hamming distance of size k. Show how the learning objective quickly becomes intractable and why in practice Monte Carlo sampling is used. Please explain your answer in terms of big-O complexity.

3 Score-Based Diffusion Models

Score-based diffusion models are a class of generative models that train by aligning the model's score function with the true data distribution's score function. The **score function** of a probability distribution $p(x)$ is defined as the gradient of the log-density, $\nabla_x \log p(x)$. In score-based models, the objective is to learn a function $s_\theta(x)$ that approximates the true score function $\nabla_x \log p_{\text{data}}(x)$.

The training objective is derived by minimizing the **Euclidean distance** between the model's score function and the true score function across the data distribution. Mathematically, this can be expressed as:

$$\mathcal{J}(\theta) = \frac{1}{2}\mathbb{E}_{p_{\text{data}}(x)}\left[\|s_\theta(x) - \nabla_x \log p_{\text{data}}(x)\|^2\right] \tag{5}$$

Through integration by parts, this objective can be simplified to a form that involves the trace of the Jacobian of the model's score function with respect to the input, along with a constant term. The simplified objective is given by:

$$\mathcal{J}(\theta) = \mathbb{E}_{p_{\text{data}}(x)}\left[\left\|\frac{1}{2}s_\theta(x)\right\|^2 + \text{Tr}\left(\nabla_x s_\theta(x)\right)\right] + C \tag{6}$$

where C is a constant.

When x is high dimensional or its trace is not explicitly available in full form, we can use the **Hutchinson's Estimator** which is an unbiased approximation of the trace of a matrix using random vectors.

More formally, for a symmetric matrix $A \in \mathbb{R}^{n \times n}$ and its trace $\text{Tr}(A) = \sum_{i=1}^n A_{ii}$, the Hutchinson Estimator can be calculated as

$$\text{Tr}(A) \approx \frac{1}{m}\sum_{j=1}^{m} z_j^\top A z_j \tag{7}$$

where $z_j \in \mathbb{R}^n$ are independent random vectors sampled from a distribution such that $\mathbb{E}[z_j z_j^\top] = I_n$ is the identity matrix.

(a) **[8 points (Coding, Extra Credit)] Implementing the Exact Score Matching**

To implement the Exact Score Matching objective, you are required to complete the following functions within the provided code in the `score_matching_utils.py` and `score_matching.py` files:

- `log_p_theta`
- `compute_score`
- `compute_l2norm_squared`
- `score_matching_objective`

Run the score matching experiment using the following command:

```
python run_score_matching.py
```

Your implementation should generate the following results:

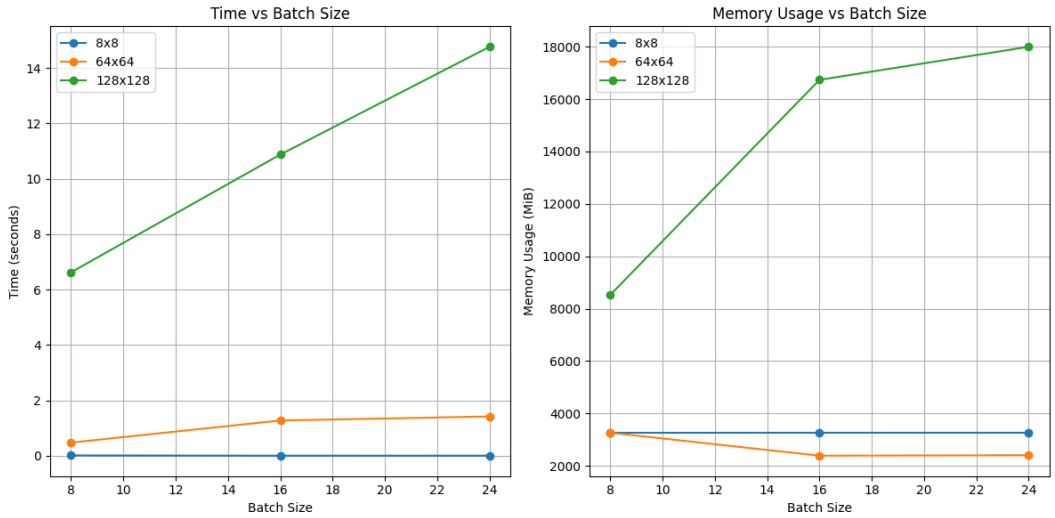

Figure 1: Score Matching Output Graphs

(b) **[2 points (Written)] Sliced Score Matching**

Given the definition of Hutchinson's Estimator, explain how it can be used to approximate the previous objective. Then derive the Sliced Score Matching Objective and explain, by providing the big-O complexity, why minimizing this objective is faster than the Exact Score Matching.

(c) **[3 points (Coding, Extra Credit)] Denoising Score Matching**

Denoising Score Matching (DSM) enhances Exact Score Matching by addressing the computational challenges associated with directly estimating the trace of the Jacobian of the score function. In Exact Score Matching, the trace of the Jacobian matrix can be computationally expensive and unstable in high dimensions. DSM avoids this by introducing Gaussian noise to the data and training the model to predict the score of the noisy data distribution. This approach indirectly aligns the score function of the model with the true data distribution without requiring explicit computation of the Jacobian.

The DSM objective is mathematically defined as:

$$\mathcal{L}_{\text{denoising}} = \mathbb{E}_{x\sim p_{\text{data}(x)}} \mathbb{E}_{z\sim\mathcal{N}(0,\sigma^2 I)} \left[\left\| \mathbf{s}_\theta(x+z) + z/\sigma^2 \right\|^2 \right], \tag{8}$$

You are required to implement the following functions in the provided code from `score_matching_utils.py` and `score_matching.py` files to make Denoising Score Matching operational:

- `add_noise`
- `compute_gaussian_score`
- `compute_target_score`
- `denoising_score_matching_objective`

Run the denoising score matching experiment using the following command:

```
python run_score_matching.py --denoise
```

Your implementation should generate the following results:

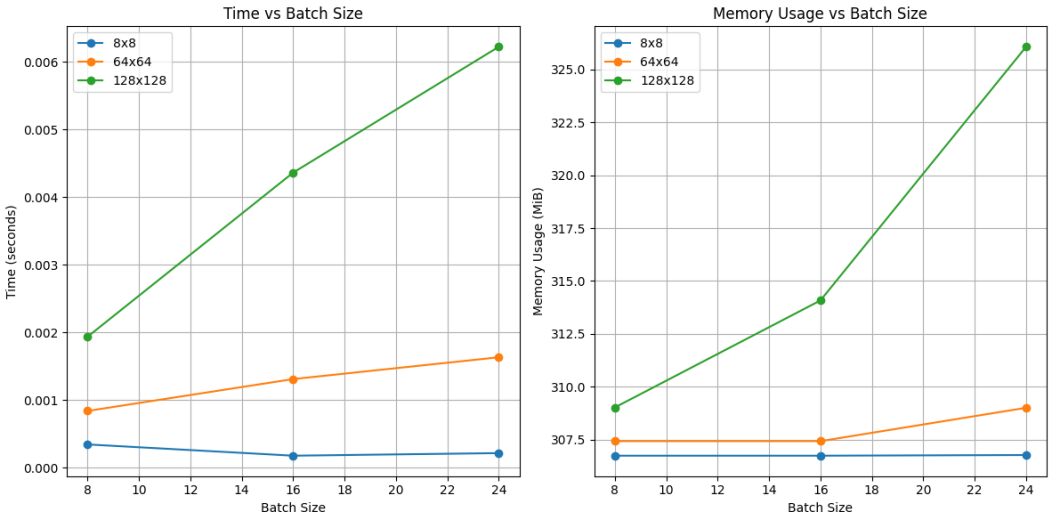

Figure 2: Denoising Score Matching Output Graphs

4 Denoising Diffusion Probabilistic Models

Diffusion models generate samples by starting from random noise and iteratively denoising it, guided by a learned model. Two common sampling approaches are:

1. **DDPM (Denoising Diffusion Probabilistic Models):** A step-by-step Markovian reverse process that typically uses many steps but ensures high-quality reconstructions.
2. **DDIM (Denoising Diffusion Implicit Models):** A non-Markovian approach allowing deterministic sampling and fewer steps, making sampling faster.

Background

Consider a forward diffusion process that gradually adds Gaussian noise to an initial image x_0 over T steps. Define a noise schedule $\{\beta_t\}$ and set $\alpha_t = 1 - \beta_t$.

Forward Process

The forward process is:

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t} x_{t-1}, (1 - \alpha_t) I).$$

After many steps, x_T is approximately isotropic Gaussian noise.

Reverse Process

We seek to reverse the forward process:

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I).$$

The model (e.g., a UNet) predicts the noise $\epsilon_\theta(x_t, t)$, from which we can estimate the original sample:

$$x_0 \approx \frac{x_t - \sqrt{1 - \bar{\alpha}_t}\, \epsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}},$$

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

(a) **[17 points (Coding)] Sampling using Vanilla DDPM and DDIM**

For DDPM, the posterior distribution of x_{t-1} given x_t and x_0 is known:

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t, \tilde{\beta}_t I),$$

with

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}} \beta_t}{1 - \bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} x_t,$$

and

$$\tilde{\beta}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t.$$

By iterating these steps backward from T to 0 and adding Gaussian noise at each step (except $t = 0$), we reconstruct a clean sample from pure noise.

Given the provided code in the `sample.py` file, please implement the following functions:

- `get_timesteps`
- `predict_x0`
- `compute_forward_posterior_mean`

- `compute_forward_posterior_variance`

To generate mnist sample run:

```
python run_sampling.py --dataset mnist --experiment ddpm
```

Run the experiment to generate few samples with the default **num_steps**=1000. **Hint:** Once we stabilize the quality at 1000, we notice that the result is always almost the same.

If you have enough time and resources you can run to generate 256x256 faces samples

```
python run_sampling.py --dataset faces --experiment ddpm
```

You will generate samples thast look like:

Figure 3: DDPM samples for Celebrity dataset

DDIM Sampling

DDIM sampling provides a shortcut:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} x_0 + \sqrt{1 - \bar{\alpha}_{t-1}} \epsilon_\theta(x_t, t).$$

This is a special case of the general formula (16) in the DDIM paper where $\eta = 0$.

Run the `ddim_inference` to implement the DDIM update following the previous equation.

To generate mnist sample run:

```
python run_sampling.py --dataset mnist --experiment ddim
```

Run the experiment to generate few samples with **num_steps**=5, 10, 20, 50. **Hint:** Once we stabilize the quality at 10, we notice that the result is always almost the same.

If you have enough time and resources you can run to generate 256x256 faces samples

```
python run_sampling.py --dataset faces --experiment ddim
```

You will now generate samples thast look like:

Figure 4: DDIM samples for Celebrity dataset

(b) **[9.50 points (Coding)] DDIM as Markovian Process**

We wish to introduce a Markovian variant of DDIM, we can incorporate a noise term at each step to make it probabilistic. Consider a parameter σ_t that controls the amount of noise injected back into the process. By doing so, we can smoothly transition from a DDIM-like deterministic sampler (when $\sigma_t = 0$) to a full stochastic, Markovian process (when $\sigma_t > 0$).

To make the previous step Markovian, we introduce a noise term with variance controlled by σ_t, which is described by the formula (16) in the DDIM paper. Let $z \sim \mathcal{N}(0, I)$ be a standard Gaussian noise sample. The Markovian variant of the DDIM update can be written as:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} x_0 + \underbrace{\sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2}\,\epsilon}_{\texttt{predict\_sample\_direction}} + \underbrace{\sigma_t z}_{\texttt{stochasticity\_term}} .$$

Please implement the following functions from the :

- `get_stochasticity_std`
- `predict_sample_direction`
- `stochasticity_term`

to accurately reproduce the update method described in the the DDIM paper by the formulas (12) and (16).

Run the experiment to generate few samples with **eta**=0, 0.2, 0.5, 0.75, 1 for **num_steps** = 10.

(c) **[2 points (Written)] DDPM vs DDIM**

What differences in sample quality and speed do you observe between DDPM and DDIM sampling?

5 Applications to Inpainting

Background

Inpainting in DDPM involves reconstructing missing regions of an image while retaining the known regions. This is achieved by blending two components:

1. The noisy image generated during the diffusion process, which provides a starting point for reconstruction.
2. The noised version of the original, uncorrupted image for the regions specified as "known" by a mask.

Task Explanation

The goal is to update the noisy image x_t at timestep t, such that:

- Known regions (indicated by a binary mask $m = 0$) are retained from x_t,
- Missing regions (indicated by $m = 1$) are replaced with the noisy version of the original image x_{orig}, created using the forward noise schedule.

This blending operation can be written as a mathematical formula involving:

- The noisy image x_t,
- The original image x_{orig},
- The binary mask m,
- A function that adds noise to x_{orig} according to the diffusion schedule, $x_{\text{orig\_noisy}}$.

The forward process to add noise can be written as

$$x_t = \sqrt{\bar{\alpha}_t}\, x_{t-1} \; + \; \sqrt{1 - \bar{\alpha}_t}\, \epsilon$$

where $\bar{\alpha}_t$ is hyperparameter constant and ϵ is the noise term.

(a) **[1 point (Written)] Inpainting Formula**

Express the inpainting update mathematically, ensuring that:

- For $m = 0$, the value is taken directly from $x_{\text{orig\_noisy}}$,
- For $m = 1$, the value is taken from the noised version of x_t.

The expression derived here will be implemented in `apply_inpainting_mask` in the next question.

(b) **[11.50 points (Coding)] Implementing Inpainting through DDPM**

From the `inpaint.py` file, implement the masking function `get_mask` by retaining only a central square of the image of side half of the image and the helper function `add_forward_tnoise`. Then update the noisy image x_t to formulate the inpainting update in the `apply_inpainting_mask` function.

Run the inpainting experiment using the following command:

```
python run_sampling.py --dataset faces --experiment inpaint --image_path /path/to/image/
to/inpaint
```

Note: For this problem we recommend using the faces dataset to see results visually. So before running the previous command one can first generate an image using ddim using the faces dataset Before running the previous command, you should first generate an image using DDIM with the faces dataset. Here's how:

```
python run_sampling.py --dataset faces --experiment ddim
```

Below is the expected result from applying inpainting by specifying the DDIM image as the image to inpaint using the `--image_path` argument:

(a) Original Image (b) Masked Image (c) Inpainted Image