# Homework 4

Welcome to Homework 4! As large-scale generative models are intractable to train by yourselves, the goal of this homework is to understand how we can *sample* from pre-trained from generation models, as well as apply a number of inference-time tricks for creative controllable generation. Here, we'll use Stable Audio Open (SAO) as our case-study pre-trained model. Note that each question in the assignment builds on previous questions, so it is recommended that you complete it **in order**.

## Setup Instructions

This homework can be done either locally, on Google Colab, or on datahub. We recommend to use colab/datahub rather than local developement, as a GPU will accelerate your ability to iterate on this assignment quickly. Note that in order to ensure compute equity, a GPU is *not* necessary for this assignment, and for this to be a case, all testing and answer generation will be done IN THE LATENT SPACE OF THE AUTOENCODER. While you are free to decode the latents into audio, we do not recommend this if you do not have a GPU, as this can be exceedingly slow on CPU.

To begin this homework, you'll need to set up your environment and obtain the necessary files. Follow these steps carefully. Ensure you have Git installed for cloning the repository and a Python environment manager (such as Anaconda) to manage dependencies.

### Local Setup

#### 1. Clone the Provided Repository

Use the following command in your terminal to download the homework files from the provided GitHub repository:

```
git clone https://github.com/ZacharyNovack/sat-253.git
```

This will create a folder named sat-253 containing all the necessary files, including the `homework4_stub.ipynb` template for the quizzes.

#### 2. Create a New Python Environment

To avoid conflicts with other projects, create a fresh Python environment. If you're using Anaconda, run these commands:

```
conda create -n hw4
conda activate hw4
```

If you prefer `virtualenv` or another tool, you can use it instead (e.g., `python -m venv hw4` and `source hw4/bin/activate` on Unix-based systems).

#### 3. Install required packages

With the `hw4` environment active, install the following packages using `pip`. These libraries are essential for completing the quizzes:

```
pip install torch torchaudio torchvision einops tqdm safetensors
matplotlib
pip install huggingface_hub einops_exts pytorch_lightning wandb
k_diffusion alias_free_torch transformers
```

Additionally, make sure you install the sat-253 directory directly as a package. In your terminal, if you are not already in the folder, run `cd sat-253`, and then run the install command:

```
pip install -e .
```

If you encounter installation errors, verify your internet connection, ensure `pip` is up-to-date (`pip install --upgrade pip`), or consult the package documentation.

#### 4. Log into Hugging face

You must sign in to Hugging Face and request access to the (Stable Audio Open)[https://huggingface.co/stabilityai/stable-audio-open-1.0] model. After that, run huggingface-cli login in your terminal:

```
huggingface-cli login
```

### Colab Setup

#### 1. Copy Repo to your personal drive

To run this on google colab, you will first need to copy the repo to your personal google drive. To do so, first download the repo `sat-253` folder (either from the class google drive or the github), and then re-upload this on your own google drive.

#### 2. Log into Hugging face

You must sign in to Hugging Face and request access to the (Stable Audio Open)[https://huggingface.co/stabilityai/stable-audio-open-1.0] model. In order to access the pre-trained weights of SAO, you must make a HuggingFace account and generate a secure token. Once you make your secure token, you will be able to directly upload it to google colab by clicking on the key icon on the left-hand side of the screen in google colab. There, you will be able to add your secret key, which should be titled `HF_TOKEN`.

#### 3. Run colab-specific codeblock

In `homework4_stub.ipynb` AND in `runner.ipynb`, there will be a colab-specific codeblock that you must run in order to set up your environment and ensure all paths work as planned. You must uncomment this block to run the notebook.

## Quizzes

This homework consists of 5 quizzes that test your ability to implement code using the provided tools and libraries. The starting template is located in the `homework4_stub.ipynb` file within the cloned repository.

- Locate and Open: Find the `homework4_stub.ipynb` file in the `sat-253` directory.
- Implement Solutions: Follow the instructions to complete the code for all 5 quizzes.
- Verify: Test your code to ensure it runs correctly and meets the requirements.
- Submit: Once finished, convert your notebook file to a `.py` file, and submit the file according to the submission instructions.

### Q1. Develope a Simple Diffusion Sampler

Diffusion models are unique among generative models in that training and inference look *drastically* different from an algorithmic perspective. As such, even given a pre-trained diffusion model, we can't repurpose its training code for generation, and instead need to design our inference algorithms, referred to as diffusion *samplers*, seperately. Here, you will implement a simple diffusion sampler, that will proceed in two subparts: (1) converting your diffusion output to the *score* of the diffusion process, and (2) using this in your sampler.

#### Q1-1. `to_d()` function

As mentioned in lecture, though diffusion models are equivalent to the class of "score matching" models (where our model directly outputs the estimated score of the distribution), they are often trained under different parameterizations, such as predicting the clean output (x-prediction), predicting the noise ($\epsilon$-prediction), or something in between (v-prediction). At a high level, you can think of the score as the *derivative* of the generation process, essentially giving you a *direction* to move your generation to make it more "data-like", and thus these gradient directions are quite important for sampling!

For our purposes, SAO uses x-prediction, which means its output is an estimate of the final denoised generation. Thus, your task is to implement the `to_d` function, which converts the x-prediction SAO output to the score.

##### Inputs

- `x`: A PyTorch tensor representing the current noisy intermediate state of the sampling process, which is the input to SAO.
- `sigma`: A scalar or tensor representing the current noise level in the diffusion process.
- `denoised`: A PyTorch tensor of the same shape as `x`, representing the model's prediction of the clean (denoised) sample at the current step.

##### Implementation Instructions

- This function should return an estimate of the score of the distribution, which is the same shape as `x` and `denoised`.

- Hint: this function should only be a 1-liner! Think about what was discussed in lecture regarding Tweedie's formula and the connection between denoising and score matching.

### Q1-2. `simple_sample()` function

Now given your `to_d` function, the goal is now to implement the simplest possible sampler for diffusion models, which we'll call `simple_sample`. Since we have a way to estimate the "direction" to move at any given noise level, sampling proceeds as follows:

1. We first start with full gaussian noise
2. We pass our noisy state into the diffusion model, along with the noise level and any extra conditioning arguments, which will return our estimate of the clean denoised output `denoised`.
3. Then, we use `to_d` to convert this to our estimate of the score.
4. Next, we calculate the **amount** that we which to step given this direction, which should be based on the current noise level and the next immediate noise level.
5. Given this stepsize, we update our noisy state using our score scaled by this stepsize.
6. Repeat steps 2-5 for some prespecified number of steps to get our final output.

#### Inputs

- `model`: The denoiser model SAO.
- `x`: A PyTorch tensor consisting of full Gaussian Noise, which will be updated throughout generation.
- `sigmas`: A list of scalar noise levels in descending order (i.e. `sigmas[i+1] < sigmas[i]`).
- `extra_args`: A dictionary of extra conditions and arguments to be passed to the model.

#### Implementation Instructions

- We have given you reasonable starter code for this function, where your main task is to implement everything inside the for loop.
- Note: The function signature of SAO is `model(x: torch.Tensor, sigma: torch.Tensor, **extra_args)`, so you'll want to follow this and scale each noise level `sigmas[i]` by the `s_in` tensor provided when passing it into the model in order to convert it from a float to a torch.Tensor.
- Like the previous question, this does not need to be particularly verbose! An efficient implementation can take as few as 4 lines of code inside the for loop.

## Q2. Implement the `generate_inpainting_mask()` function.

Now that we have a our `simple_sample` function, we can modify it in a number of ways for creative, training-free control. First, we'll explore **inpainting**, or the task of taking existing audio, masking out some portion of the middle, and then filling it in with our model! First off, we need to make a helper function to generate these masks.

Build the `generate_inpainting_mask()` function, which creates an inpainting mask for audio data, specifying regions of an audo waveform to be "inpainted".

### Inputs

- `reference`: A PyTorch tensor representing the reference **latent** audio to be inpainted.
- `mask_start_s`: A float representing the start time of the mask in *seconds*

- `mask_end_s`: A float representing the end time of the mask in *seconds*

#### Constants Provided

- `SAMPLE_RATE` : A global constant representing the audio sample rate
- `model.pretransform.downsampling_ratio`: This is the factor that the audio is downsampled through SAOs encoder to its latent representation. We need this as all our masking occurs in the the latent space, so we need to account for the latent frame rate of SAO's autoencoder.

#### Implementation Instructions

- your function should return a mask `mask` that is the same shape as `reference`, where `mask=1` everywhere that we want to inpaint (i.e. from `mask_start_s` to `mask_end_s`) and `mask=0` everywhere else.
- you will need to combine `mask_start_s`/`mask_end_s` with `SAMPLE_RATE` and `model.pretransform.downsampling_ratio` to set the correct indices of the mask.

### Q3. Implement `simple_sample_inpaint()` function.

Next, we'll implement inpainting. The idea here is that we can reuse our simple sampler with some small modifications. Notably, after we do the sampling step and update our noisy state, we want to do two things:

1. Make sure that we scale whatever is coming from our reference to the same noise level, which is equivalent to adding a specific amount of gaussian noise at each step
2. Given our synthetically noisy reference, we want to use the mask to set `x` such that in the inpainting range `x` stays unchanged, but outside the range `x` is set to our noisy reference.

The `simple_sample_inpaint()` function is the implementation of the simple sampling method for inpainting.

#### Inputs

- `model`: The stable audio open model that we're using
- `x`: A PyTorch tensor representing the initial noisy sample
- `sigmas`: A PyTorch tensor of noise levels
- `reference`: The encoded reference audio.
- `mask`: The mask that we created with the `generate_inpainting_mask` function.

#### Implementation Instructions

- Note: this is again a small modification on your existing simple sample code, do not overthink it!
- Think about what it means to add noise at the "right" noise level, and how to do a mask operation.

### Q4. Variable Strength Impainting

We now have a working inpainting function! As we can see, when playing around with different prompts and reference audios, the outut around the boundaries of our inpainting mask can feel abrupt and unnatural. To potentially fix this, next you will modify your inpainting function to only apply the inpainting operation on a specific *range* of steps.

The `simple_sample_variable_inpaint()` function is the implementation of the simple sampling method for inpainting, with variable inpainting amount.

**Inputs**

- `model`: The stable audio open model that we're using
- `x`: A PyTorch tensor representing the initial noisy sample
- `sigmas`: A PyTorch tensor of noise levels
- `reference`: The encoded reference audio.
- `mask`: The mask that we created with the `generate_inpainting_mask` function.
- `paint_start`: An integer index denoting the step number to start apply the inpainting function.
- `paint_end`: An integer index denoting the step number to stop apply the inpainting function.

**Implementation Instructions**

- Note: this is again a small modification on your existing simple sample inpaint code, do not overthink it!
- This function should operate identically to your simple sample inpaint function when `paint_start=0` and `paint_end=len(sigmas)-1` (i.e. applying the inpainting operation for all of sampling).

### Q5. Style Transfer

Finally, we will explore a simple version of style transfer with diffusion models. The goal here is to "transfer" the style from one reference audio to our generation. This ends up being very similar to inpainting, with the exception that this is a *global* operation (i.e. no mask is needed), and is more concerned with the *strength* of the transfer operation. Specifically, we want to perform style transfer by taking the reference audio, adding noise to it up to some level, and then running inference from this noisy reference point as normal.

The `simple_sample_style_transfer()` function is the implementation of the simple sampling method for style transfer.

**Inputs**

- `model`: The stable audio open model that we're using
- `sigmas`: A PyTorch tensor of noise levels
- `reference`: The encoded reference audio.
- `transfer_strength`: A float value from 0 to 1, where 0 means no transfer occurs (i.e. the generation ignores the reference) and 1 means maximum transfer (i.e. the output should just be the reference).

**Implementation Instructions**

- There are a couple ways to implement this, either using your `simple_sample_variable_inpaint` or your `simple_sample` function.
- You will need to conver your transfer_strength from a 0-1 float to something that can index the sigmas tensor to map it to the right noise level.

## Submission Instructions

To submit your homework, first convert your `homework4_stub.ipynb` file into a `homework4_stub.py` file. NOTE: before converting your notebook into a python file, MAKE SURE TO COMMENT OUT ALL `ipd` CALLS IN YOUR FILE, as this will break the importing to the answer generation file. Additionally, it will be helpful to comment out all testing blocks in your code before you use the runner, as these will slow down the import process considerably. Then navigate to the `runner.ipynb` file (in the sat-253 directory) and run the whole notebook, which should import your written functions from `homework4_stub.py` and generate an `homework4.pkl` file. Submit both this file, and your `homework4_stub.py` file on gradescope.