Diffusion Models for Inverse Problems

M2 MVA course, Deep Learning for Image Restoration & Synthesis

Authors: Charles Laroche & Andrés Almansa

I) Setup

```
# Set home directory containing this notebook + data

# HOMEDIR = "/Users/almansa/Home/Docencia/DeLIReS/2024/4-PosteriorSampling/code/TP-DM"
HOMEDIR = "/content/drive/MyDrive/restauration_image/TP3/TP-DM"
```

Mount Google Drive

+ Code + Text

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Go to the folder in Google drive where you downloaded the provided code. And check its contents.

```
%cd /content/drive/MyDrive/restauration_image/TP3/TP-DM
!ls -l
```

```
/content/drive/MyDrive/restauration_image/TP3/TP-DM
total 2232
drwx------ 5 root root    4096 Feb 11 13:50 data
drwx------ 2 root root    4096 Feb 11 13:50 images_notebook
drwx------ 2 root root    4096 Feb 11 13:50 __MACOSX
-rw------- 1 root root     619 Jan 28 02:16 README.md
-rw------- 1 root root      43 Sep 22 15:02 requirement.txt
-rw------- 1 root root   37820 Jan 30 15:12 TP-DM.ipynb
-rw------- 1 root root   25113 Jan 30 03:19 TP-DM.md
-rw------- 1 root root 2203329 Jan 30 03:19 TP-DM.pdf
drwx------ 3 root root    4096 Feb 11 13:50 utils
```

Select GPU / CPU

By the end of this section `device` should point to a cuda or mps GPU if one is available, otherwise it should point to a CPU.

For google colab: If a cuda GPU is not available, change the runtime type in the Runtime menu.

```
!nvidia-smi
```

```
Sun Feb 11 21:49:29 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   47C    P8              10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

The following code checks if Apple Silicon or nvidia GPU is available. In that case it sets `device` to "mps" or "cuda" respectively. Otherwise fallback to `device="cpu"`. If you have access to a TPU in google colab or another non-cuda GPU you may need to change the code below.

```
# Select best GPU, fallback to CPU if no GPU is available
import os, torch

# If Apple Silicon processor is available set device to "mps"
if torch.backends.mps.is_available():
    device = torch.device("mps")
# If nvidia GPU is available set device to "cuda"
elif torch.cuda.is_available():
    device = torch.device("cuda")
# otherwise fallback to "cpu"
else:
    device = torch.device("cpu")
    print ("GPU not found using CPU.")

# device = torch.device("cpu")

device
```

```
device(type='cuda')
```

Install required packages

```
!pip install diffusers accelerate #PACKAGE NEEDED FOR COLAB
#DIFFFUSER lib from higging face allows to train pretreaiend models
```

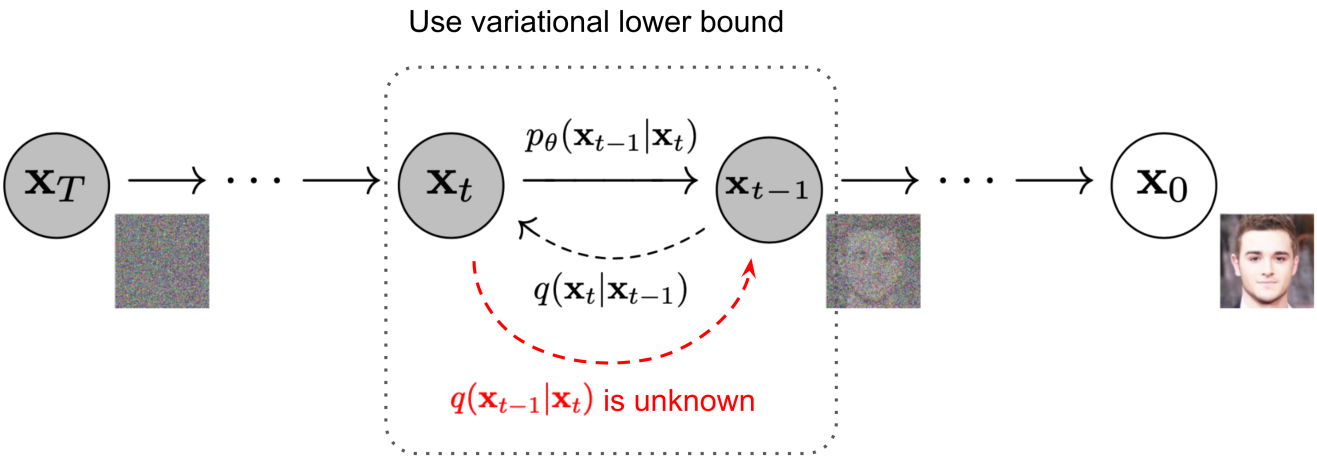
```
Requirement already satisfied: diffusers in /usr/local/lib/python3.10/dist-packages (0.26.2)
Requirement already satisfied: accelerate in /usr/local/lib/python3.10/dist-packages (0.27.0)
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.10/dist-packages (from diffusers) (7.0.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from diffusers) (3.13.1)
Requirement already satisfied: huggingface-hub>=0.20.2 in /usr/local/lib/python3.10/dist-packages (from diffusers) (0.20.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from diffusers) (1.23.5)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from diffusers) (2023.12.25)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from diffusers) (2.31.0)
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10/dist-packages (from diffusers) (0.4.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from diffusers) (9.4.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from accelerate) (23.2)
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from accelerate) (5.9.5)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from accelerate) (6.0.1)
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/dist-packages (from accelerate) (2.1.0+cu121)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.20.2->diffusers) (20
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.20.2->diffusers) (4.66.1
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.20.2->diff
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate) (3.1.3)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate) (2.1.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packages (from importlib-metadata->diffusers) (3.17.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->diffusers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->diffusers) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->diffusers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->diffusers) (2024.2.2)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.10.0->accelerate) (2.1.
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.10.0->accelerate) (1.3.0)
```

Import required libraries

```
import matplotlib.pyplot as plt
import torch
import tqdm
import utils.utils_image as util
import utils.utils_agem as agem
```

II) DDPM: Denoising Diffusion Probabilistic Models

Introduction



Denoising diffusion probabilistic models (DDPM) can learn to sample from a distribution of natural images $p(x_0)$.

To do so, they first define a forward process that progressively adds Gaussian noise to x_0 until $x_T \sim \mathcal{N}(0, I)$:

$$
\begin{aligned}
&q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \\
&q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t Id) \\
&x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}e_t \quad \text{where} \quad e_t \sim \mathcal{N}(0, Id) \\
&x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t \quad \text{where} \quad \epsilon_t \sim \mathcal{N}(0, Id), \;\; \bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)
\end{aligned}
$$

In order to sample from $q(x_0)$ we need to estimate the reverse process $q(x_{t-1}|x_t)$.

According to (Feller 1949; Sohl-Dickstein et al 2015), if β_t is infinitesimally small then $q(x_{t-1}|x_t)$ is also Gaussian, so we can approximate it by

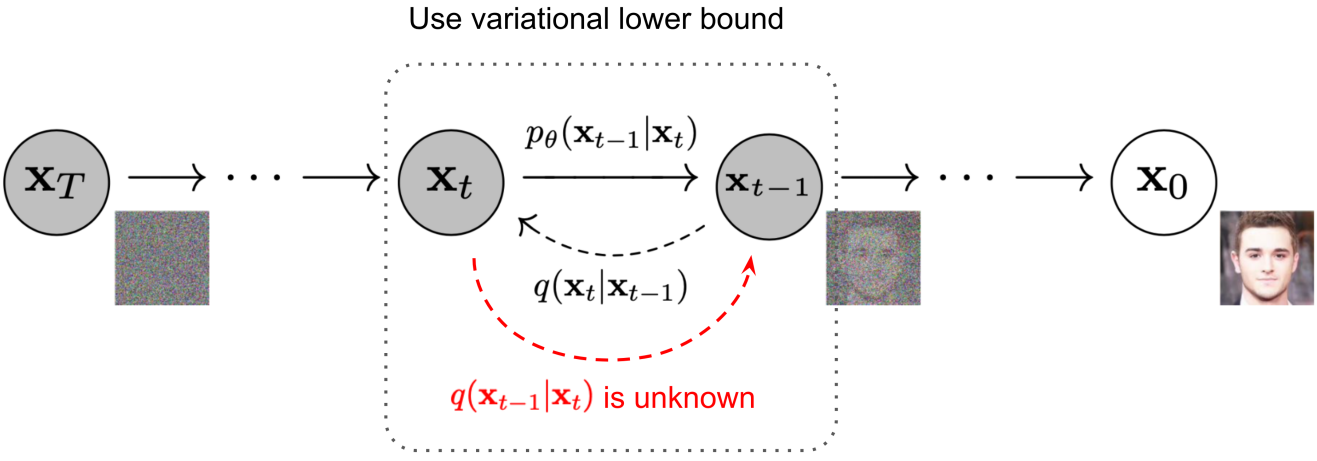
$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_t; \mu_\theta(x_t, t), \sigma_t^2 I)$$

Using variational inference tools (Ho et al 2020) conclude that

- $\mu_\theta(x_t, t) = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t, t)\right)$
- $\sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$

where $\hat{\epsilon}_\theta(x_t, t)$ is a neural network (denoiser) that is trained to predict the noise ϵ_t in $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$.

The DDPM sampling algorithm is summarized as follows



Denoising diffusion probabilistic models (DDPM) can learn to sample from a distribution of natural images $p(x_0)$.

To do so, they first define a forward process that progressively adds Gaussian noise to x_0 until $x_T \sim \mathcal{N}(0, I)$:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$
$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t Id)$$
$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}e_t \quad \text{where} \quad e_t \sim \mathcal{N}(0, Id)$$
$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t \quad \text{where} \quad \epsilon_t \sim \mathcal{N}(0, Id), \;\; \bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$$

In order to sample from $q(x_0)$ we need to estimate the reverse process $q(x_{t-1}|x_t)$.

According to (Feller 1949; Sohl-Dickstein et al 2015), if β_t is infinitesimally small then $q(x_{t-1}|x_t)$ is also Gaussian, so we can approximate it by

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

Using variational inference tools (Ho et al 2020) conclude that

- $\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t, t)\right)$
- $\sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$

where $\hat{\epsilon}_\theta(x_t, t)$ is a neural network (denoiser) that is trained to predict the noise ϵ_t in $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$.

Questions

In the sequel we shall need to establish a link between

- the noise estimator $\hat{\epsilon}_\theta$ and
- the score function $\nabla \log p(x_t)$.

This is the goal of the following questions:

> **Question 1:** Use the definition of the forward process $x_t \sim \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, \, (1-\bar{\alpha}_t)I)$ and Tweedie's identity to show that $\widehat{x_0}(x_t) = E[x_0|x_t] = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t + (1-\bar{\alpha}_t)\nabla \log p(x_t))$.

We have:

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$$

Tweedie's identity is gven by:

$$\frac{1}{\sigma^2}(D_\sigma^* - I)(x) = \nabla \log p_\sigma(x)$$

Therefore is this case the variance is: $\sigma^2 = (1-\bar{\alpha}_t)$ Then given that $D_\sigma^{MMSE}(x_t) = E[x_0|x_t]$

We replace in the previous formula and get:

$$\frac{1}{(1-\bar{\alpha}_t)}(E[x_0|x_t] - x_t) = \nabla \log p(x_t)$$
$$E[x_0|x_t] = x_t + (1-\bar{\alpha}_t)\nabla \log p(x_t)$$

Finally the estimate is up to the scaling factor $\frac{1}{\sqrt{\bar{\alpha}_t}}$:

$$\hat{x}_0(x_t) = E[x_0|x_t] = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t + (1-\bar{\alpha}_t)\nabla \log p(x_t))$$

> **Question 2:** Use the defition of the forward process $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\,\epsilon$ to find an expression of x_0 in terms of ϵ

we get:

$$x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_t)$$

> **Question 3:** Conclude from the two previous results that
> $\nabla \log p(x_t) = -\frac{1}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t)$

We know that $\hat{\epsilon}_\theta(x_t)$ is trained to approximate ϵ_t and that $\hat{x}_0(x_t)$ is the estimator of x_0, we replace these quantities in the result of Question 2:

$$\hat{x}_0(x_t) = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}_\theta(x_t))$$

Also from question 1, we get:

$$\nabla \log p(x_t) = \frac{1}{(1-\bar{\alpha}_t)}\left(\sqrt{\bar{\alpha}_t}\hat{x}_0 - x_t\right)$$

Replacing $\hat{x}_0$ by the output of the previous expression we get:

$$\nabla \log p(x_t) = -\frac{1}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t)$$

> **Question 4:** Use the previous result to show that the unconditional DDPM update rule can be written in terms of the score as
> $$x'_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t + (1-\alpha_t)\nabla \log p(x_t)) + \sigma_t z_t$$

We have: $p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_t; \mu_\theta(x_t,t), \sigma_t^2 I)$. we use the expressions of $\mu_\theta(x_t,t)$ and σ_t^2 given above to write:

$$x'_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t,t)\right) + \sigma_t z_t \quad \text{where} \quad z_t \sim \mathcal{N}(0, Id)$$

Also from obtained: $-\frac{1}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_\theta(x_t) = \nabla \log p(x_t)$

We replace by its expression to get:

$$x'_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t + (1-\alpha_t)\nabla \log p(x_t)) + \sigma_t z_t$$

Using the pipeline

Now we are ready to load a pretrained DDPM model and use it to sample from the distribution of face images on which it was trained.

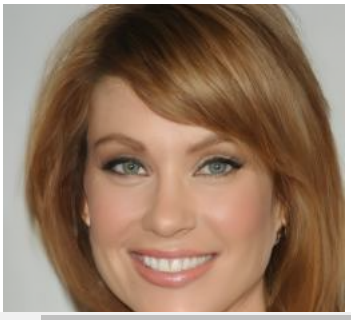
To do so we use the diffusers library from HuggingFace.

```
from diffusers import DDPMPipeline

# Load pretrained model from https://huggingface.co/models?sort=downloads&search=ddpm
model_name="google/ddpm-ema-celebahq-256" # 256x256 8 iteratons/second

ddpm = DDPMPipeline.from_pretrained(model_name).to(device)
image = ddpm(num_inference_steps=100).images[0]
image

###model entraine su 1000 etapes pour generrer de bonned images
### this very high level just download the model and apply it for t=100
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secre
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
diffusion_pytorch_model.safetensors not found
```

Loading pipeline components...: 100% 2/2 [00:01<00:00, 1.09it/s]

100% 100/100 [00:11<00:00, 8.72it/s]



What's behind

The pipeline is composed of two main elements:

- `unet` is the pretrained score model that is used to predict x_0 from x_t and t.
- `scheduler` contains the values β_t, α_t and $\bar{\alpha}_t$, as well as the function to predict x_{t-1} from x_t and t.

for a closer look

we have a une that will predict epsilon from xt and scheluder who will go from xt to xt-1

```
# Either load pipeline from pretrained models
ddpm = DDPMPipeline.from_pretrained(model_name).to(device)
# and then extract scheduler and model
scheduler = ddpm.scheduler
model = ddpm.unet

# Or load scheduler and model directly
# from diffusers import DDPMScheduler, UNet2DModel
# scheduler = DDPMScheduler.from_pretrained(model_name)
# model = UNet2DModel.from_pretrained(model_name).to(device)

# You can also try a different scheduler
# from diffusers import DDIMScheduler, UNet2DModel
# scheduler = DDIMScheduler.from_pretrained(model_name)
# model = UNet2DModel.from_pretrained(model_name).to(device)

scheduler.set_timesteps(100)
```

```
diffusion_pytorch_model.safetensors not found
Loading pipeline components...: 100%  2/2 [00:00<00:00, 2.07it/s]
```

```
print(ddpm.scheduler)
```

```
DDPMScheduler {
  "_class_name": "DDPMScheduler",
  "_diffusers_version": "0.26.2",
  "beta_end": 0.02,
  "beta_schedule": "linear",
  "beta_start": 0.0001,
  "clip_sample": true,
  "clip_sample_range": 1.0,
  "dynamic_thresholding_ratio": 0.995,
  "num_train_timesteps": 1000,
  "prediction_type": "epsilon",
  "rescale_betas_zero_snr": false,
  "sample_max_value": 1.0,
  "steps_offset": 0,
  "thresholding": false,
  "timestep_spacing": "leading",
  "trained_betas": null,
  "variance_type": "fixed_small"
}
```

Next we shall use the scheduler and the model to write a lower level version of the DDPM sampling algorithm.

Before that,

- let's have a look at the scheduler (this requires somme utility functions below)
- and let's have a look at the UNet model.

Utility functions

```
def scheduler_alpha_t(scheduler, t): #gets the values of aplha and beta  from alpa_bat but we need to have alpha t and alpha t-1
    # Get \bar\alpha_t and \alpha_t from scheduler
    #   t should be an integer between 0 and T
    #   T = scheduler.num_train_timesteps # 1000
    prev_t = scheduler.previous_timestep(t)
    alpha_prod_t = scheduler.alphas_cumprod[t]
    alpha_prod_t_prev = scheduler.alphas_cumprod[prev_t] if prev_t >= 0 else scheduler.one
    #beta_prod_t = 1 - alpha_prod_t
    #beta_prod_t_prev = 1 - alpha_prod_t_prev
    current_alpha_t = alpha_prod_t / alpha_prod_t_prev
    #current_beta_t = 1 - current_alpha_t
    return alpha_prod_t, current_alpha_t

#display dofferent values of alpa bar
def print_scheduler(scheduler, T, stride):
    #T = scheduler.num_train_timesteps # 1000
    print("T", "t", "eta_t", "alpha_prod_t", "beta_prod_t", "current_beta_t")
    for t in range(0,T,stride):
        alpha_prod_t, current_alpha_t = scheduler_alpha_t(scheduler, t)
        beta_prod_t = 1 - alpha_prod_t
        current_beta_t = 1 - current_alpha_t
        eta_t = current_beta_t / torch.sqrt(current_alpha_t)
        print(T, t, eta_t, alpha_prod_t, beta_prod_t, current_beta_t)
#put everything in tensor
def scheduler_alphas(scheduler):
    T = scheduler.num_inference_steps # 1000
    stride=1
    n = T #math.ceil((T+1)/stride)
    alpha = torch.zeros(n)
    alpha_bar = torch.zeros(n)
    time = torch.zeros(n)
    for t in range(0,T,stride):
        alpha_prod_t, current_alpha_t = scheduler_alpha_t(scheduler, t)
        #beta_prod_t = 1 - alpha_prod_t
        #current_beta_t = 1 - current_alpha_t
        #bt = current_beta_t / torch.sqrt(current_alpha_t)
        alpha[t] = current_alpha_t
        alpha_bar[t] = alpha_prod_t
        time[t] = t
    return alpha, alpha_bar, time
```

Explore the scheduler

The variances β_t of the VP (Ornstein Uhlenbeck) forward diffusion process are chosen such that:

- $\bar{\alpha}_0$ = 1
- $\bar{\alpha}_T = 0$

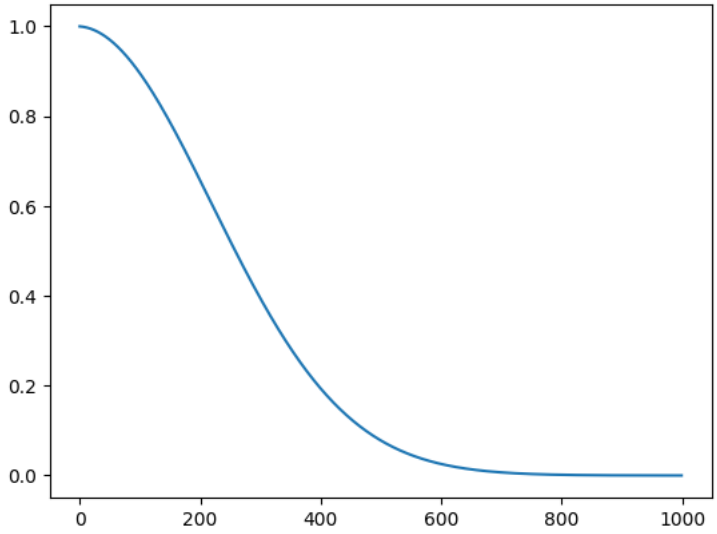
Let's plot the values of $\bar{\alpha}_t$ for $t \in [0, T]$.

2 number of times steps, the one used in learning or the one used in inference that can't go beyon 1000

le schduler recupere du model prentraine alwyas go together modele t scheduler they always come together variant DDIM

```
# T = number of timesteps during training
T = scheduler.num_train_timesteps # 1000

# TI = number of timesteps during inference
TI = T
scheduler.set_timesteps(TI)
print(scheduler.num_inference_steps)

# Get variance schedule
alpha, alpha_bar, time = scheduler_alphas(scheduler)

# Plot variance schedule
plt.plot(time, alpha_bar, label="alpha_bar")

# Print variance schedule
print_scheduler(scheduler, TI, int(TI/10))
```

```
1000
T t eta_t alpha_prod_t beta_prod_t current_beta_t
1000 0 tensor(0.0001) tensor(0.9999) tensor(0.0001) tensor(0.0001)
1000 100 tensor(0.0021) tensor(0.8951) tensor(0.1049) tensor(0.0021)
1000 200 tensor(0.0041) tensor(0.6563) tensor(0.3437) tensor(0.0041)
1000 300 tensor(0.0061) tensor(0.3940) tensor(0.6060) tensor(0.0061)
1000 400 tensor(0.0081) tensor(0.1936) tensor(0.8064) tensor(0.0081)
1000 500 tensor(0.0101) tensor(0.0778) tensor(0.9222) tensor(0.0101)
1000 600 tensor(0.0121) tensor(0.0256) tensor(0.9744) tensor(0.0121)
1000 700 tensor(0.0141) tensor(0.0069) tensor(0.9931) tensor(0.0140)
1000 800 tensor(0.0162) tensor(0.0015) tensor(0.9985) tensor(0.0160)
1000 900 tensor(0.0182) tensor(0.0003) tensor(0.9997) tensor(0.0180)
/usr/local/lib/python3.10/dist-packages/diffusers/configuration_utils.py:139: FutureWarning: Accessing config attribute `num_train_times
  deprecate("direct config name access", "1.0.0", deprecation_message, standard_warn=False)
```



on voit la fornmule cos qui aete utilisé pour alpha pour aller de 1 a 0, and we clearly see the 1000 STEPS we generally start at alpha bar=zero only noise and the noise decreases .. we need beta t to get his profole of alpha t

DDPM Sampling in detail

Each iteration of the DDPM algorithm is composed of two steps:

1. Use the `model` to predict the noise from x_t

$$\hat{\epsilon}_t = \epsilon_\theta(x_t, t)$$

2. Use the `scheduler` to sample

$$x_{t-1} \sim p_\theta(x_{t-1}|x_t)$$

:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}\right) + \mathcal{N}(0, \sigma_t^2 I)$$

```
def ddpm_sampling(model, scheduler, n_samples=1):
    # Get hyper_params
    sample_size = model.config.sample_size

    # Init random noise
    x_T = torch.randn((n_samples, 3, sample_size, sample_size)).to(device)
    x_t = x_T

    for t in tqdm.tqdm(scheduler.timesteps):
        with torch.no_grad():
            # Get noisy residual prediction
            epsilon_t = model(x_t, t).sample # va calculer l'estimeteru de epsilon from xt-1

            # Sample x_{t-1}|x_t
            x_t = scheduler.step(epsilon_t, t, x_t).prev_sample

    # Normalize output
    x_0 = (x_t / 2 + 0.5).clamp(0, 1)
    return x_0


scheduler.set_timesteps(200)#! to change time steps
res = ddpm_sampling(model, scheduler) #par defaut il fait 1000 pas
plt.figure(figsize=(5, 5))
plt.imshow(util.tensor2uint(res))
plt.axis('off')
plt.show()
```

```
100%|██████████| 200/200 [00:23<00:00,  8.67it/s]
```

To save time you can also select a smaller number of steps.

```
TI = 100
scheduler.set_timesteps(TI)
res = ddpm_sampling(model, scheduler)
plt.figure(figsize=(5, 5))
plt.imshow(util.tensor2uint(res))
plt.axis('off')
plt.show()
```

```
100%|██████████| 100/100 [00:11<00:00,  8.58it/s]
```

Faire l'implementation de DPZ

III) Diffusion models for inverse problems

Introduction

Consider an inverse problem with Gaussian likelihood

$$p(y|x) = \mathcal{N}(Hx, \sigma^2 I)$$

and prior $p(x)$ given by a diffusion model.

Recall from Question 4 that the DDPM update rule can be written in terms of the score function as

$$x'_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t + \beta_t \nabla \log p(x_t)) + \sigma_t z_t \qquad (1)$$

For inverse problems we target the posterior $p(x|y)$ instead of the prior $p(x)$. This means that the score function in the previous equation should be substituted by

$$\nabla_{x_t} \log p(x_t|y) = \nabla_{x_t} \log p_t(x_t) + \nabla_{x_t} \log p_t(y|x_t)$$

This leads to the following DDPM update rule for posterior sampling

$$\begin{aligned} x_{t-1} &= \frac{1}{\sqrt{\alpha_t}}(x_t + \beta_t(\nabla \log p(x_t) + \nabla \log p(y|x_t))) + \sigma_t z_t \qquad (2) \\ &= \underbrace{\frac{1}{\sqrt{\alpha_t}}(x_t + \beta_t \nabla \log p(x_t)) + \sigma_t z_t}_{=x'_{t-1}} + \frac{\beta_t}{\sqrt{\alpha_t}} \nabla \log p(y|x_t) \end{aligned}$$

All the terms in the previous equation are known except for $p_t(y|x_t)$.

But we do know $p_t(y|x_0)$.

In addition the link between x_0 and x_t is provided by the forward and backward process.

Therefore we could compute the unknown term by marginalization:

$$p(y|x_t) = \int p(x_0|x_t) p(y|x_0) dx_0 \qquad (3)$$

The computation of this integral is intractable in general. But it can be computed in closed form if we adopt an approximation of $p(x_0|x_t)$.

In the next section, we present two different approximations that provide two conditional sampling algorithms from an unconditional diffusion model: DPS and ΠGDM.

Get a sample from the dataset

The dataset below contains:

- 20 images x of size 256x256 (from FFHQ face image database)
- several blur kernels k of size 33x33 (simulated camera shake kernels)

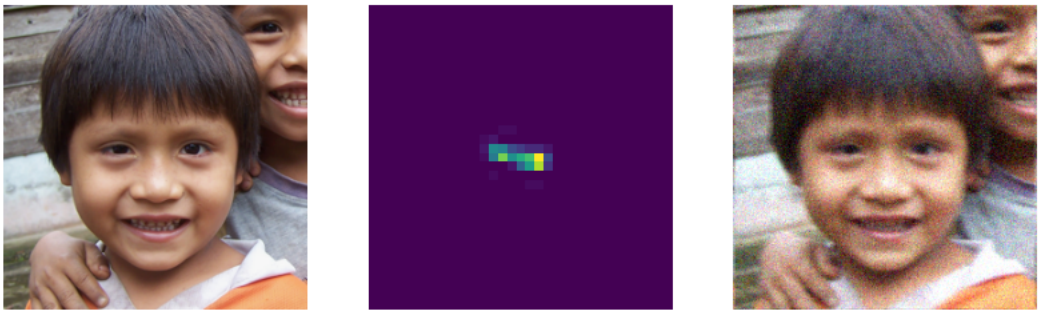
From a given pair (x, k), it generates a blurred and noisy image

$$y = \underbrace{k * x}_{=Hx} + \mathcal{N}(0, \sigma^2 Id)$$

-----(1) donne le scire et (2) donne le scheduler

----faire des convoloution

```
from data.data_blur import Dataset
from numpy import random

data_opt = {'dataroot_H': HOMEDIR + '/data/FFHQ/GT/H',
            'dataroot_kernels': HOMEDIR + '/data/kernels/custom_blur_centered.mat',
            'sigma': [1,2]}
#on utilise des flous qui simulent la camera shape
#de base on connait l'operateur h et on eut l'inverser
data = Dataset(data_opt)

# Get a sample from the dataset
k = random.randint(0, 19)
sample = data[k]
x = sample['H'] # original (ground truth) image - image k
k = sample['kernel'] # blur kernel - selected randomly (independently from k) #nous donne le noyeau
y = sample['L'] # degraded (blurred + noisy) image y = k*x + n


plt.figure(figsize=(10, 10/3))
plt.subplot(131)
plt.imshow(util.tensor2uint(x))
plt.axis('off')
plt.subplot(132)
plt.imshow(util.tensor2uint(k))
plt.axis('off')
plt.subplot(133)
plt.imshow(util.tensor2uint(y))
plt.axis('off')
plt.show()

#au milieux c'estle noyau de flou
```



Diffusion by Posterior Sampling (DPS)

------on a deja des fonction pour cacluer le forward model

DPS approximates $p(x_0|x_t)$ by a Dirac centered at the posterior mean $\widehat{x_0}(x_t) = E[x_0|x_t]$.

In other words, $p(x_0|x_t) \approx \delta_{\widehat{x_0}(x_t)}(x_0)$.

Substituting this approximation in equation (3) we obtain:

$$-\nabla_{x_t} \log p(y|x_t) = \nabla_{x_t} \frac{1}{2\sigma^2} \|H\widehat{x_0}(x_t) - y\|_2^2 =: g$$

This guidance term g can be computed using automatic differentiation

> **Question 5:** Assume that the torch tensor `f` holds the expression
> $$-\log p(y|x_t) + C = \frac{1}{2\sigma^2} \|H\widehat{x_0}(x_t) - y\|_2^2 =: f.$$
> Use `torch.autograd.grad` to compute $\nabla_{x_t} f$. Complete the DPS code below.

Combining this with equation (2) above we obtain the **DPS update rule**

$$x_{t-1} = x'_{t-1} - \eta_t g$$

where

- x'_{t-1} is the update rule for the unconditional DDPM sampler,
- g is the guidance term to make it conditional (on the observation y), and
- η_t is a guidance weight.

> **Question 6:** According to equation (2) what should be the value of η_t ? Complete the DPS code below.

From equation (2) we get the value of η_t as follows:

$$\eta_t = \frac{\beta_t}{\sqrt{\alpha_t}}$$

The authors of DPS prefer to set $\eta_t = 1$ for all t instead of the theoretical value below. You shall compare both approaches below.

> **Question 7:** Assume that the likelihod is not Gaussian, but governed by some Gibbs density $\log p(y|x_0) + C = \mathcal{A}(x_0)$. Can you still apply DPS if $\mathcal{A}$ is differentiable?

Yes as long as $\mathcal{A}(x_0)$ is differentiable we can apply DPS, we just replace $\nabla_{x_t} \log p(y/x_t)$ by the gradient $\nabla_{x_t} \mathcal{A}(x_0)$

> **Question 8:** Compare the results of DPS with fixed and adaptive η_t with ΠGDM below.

from a qualitative point of view we can see that the resulting image from adaptive η_t with ΠGDM is better quality than the one generated using a fixed η_t, and this is verified when looking at the value of the snr which was: 6.20 and 15.21 for bth methods respectively fro only a 100 steps. This implies that dynamically adjusting the guidance weight η_t to the diffusion timestep significantly enhances the fidelity of the reconstructed image.

Reference:

- DPS: https://openreview.net/forum?id=OnD9zGAGT0k

```
import utils.utils_agem as agem

def alpha_beta(scheduler, t):
    prev_t = scheduler.previous_timestep(t)
    alpha_prod_t = scheduler.alphas_cumprod[t]
    alpha_prod_t_prev = scheduler.alphas_cumprod[prev_t] if prev_t >= 0 else scheduler.one
    current_alpha_t = alpha_prod_t / alpha_prod_t_prev
    current_beta_t = 1 - current_alpha_t
    return current_alpha_t, current_beta_t


# DPS with DDPM and intrinsic scale
def dps_sampling(model, scheduler, y, fowrard_model, nsamples=1, scale=1, scale_guidance=1):
    sample_size = model.config.sample_size

    # Init random noise
    x_T = torch.randn((nsamples, 3, sample_size, sample_size)).to(device)
    x_t = x_T

    for t in tqdm.tqdm(scheduler.timesteps):

        # Predict noisy residual eps_theta(x_t)
        x_t.requires_grad_()
        epsilon_t = model(x_t, t).sample

        # Get x0_hat and unconditional
        # x_{t-1} = a_t * x_t + b_t * epsilon(x_t) + sigma_t z_t
        # with b_t = eta_t
        predict = scheduler.step(epsilon_t, t, x_t)
```

```
        x0_hat  = agem.clean_output(predict.pred_original_sample)
        x_prev  = predict.prev_sample # unconditional DDPM sample x_{t-1}'

        # Guidance
        f = torch.norm(forward_model(x0_hat) - y)
        g = -torch.autograd.grad(f, x_t, create_graph=True)[0]   # ... COMPLETE THIS LINE

        # compute variance schedule
        alpha_t, beta_t= alpha_beta(scheduler, t)

        # Guidance weight
        # eta_t = ...
        if (scale_guidance==1):
            eta_t = beta_t/ torch.sqrt(alpha_t)  # ... COMPLETE THIS LINE

        else:
            eta_t = 1.0


        # DPS update rule = DDPM update rule + guidance
        x_t = x_prev - scale * eta_t * g # parametre de scale pour multiplier eta t
        x_t = x_t.detach_()

    return agem.clean_output(x_t)
  #a la fin x_prev + qlq chose qui introduit le terme de bruit
```

Fixed $\eta_t = 1$

Now we are ready to run the DPS algorithm with fixed $\eta_t = 1$.

For best results you should use T=1000 steps. (about 5 minutes)

For faster results you can use T=100 steps. (about 30 seconds)

---- etat t a camauler from beta et alpha tester avec eta =1 et eta theorique ----- fait 4 iteratoin par secondes au leiu de 2 -------- astuce: pour quelqques fonction qui ne fonctinne pas sur gpu mais pas de probeleme sur colab

-------- la fonction forward on la passe en paramere a dps (((val juste avant))))

--- en gros c'est juste pour eta+1 mais faut faire avec 100 iterations

---

```
# torch.roll is not supported for MPS device, so we need to enable CPU fallback
%env PYTORCH_ENABLE_MPS_FALLBACK=1

# scheduler = DDPMScheduler.from_pretrained(model_name)
# model = UNet2DModel.from_pretrained(model_name).to(device)

ddpm = DDPMPipeline.from_pretrained(model_name).to(device)
scheduler = ddpm.scheduler
model = ddpm.unet

T = 100
scheduler.set_timesteps(T)
print_scheduler(scheduler, T, 10)


# Forward model (cuda GPU implementation)
# forward_model = lambda x: agem.fft_blur(x, sample['kernel'].to(device))
# If you are using an MPS gpu use the following forward_model instead
# CPU fallback implementation (no MPS support for torch.roll, fft2, Complex Float, etc.)
forward_model_cpu = lambda x: agem.fft_blur(x, sample['kernel'])
forward_model = lambda x: forward_model_cpu(x.to('cpu')).to(device)
# Degraded image y = A x + noise
y = sample['L'].to(device)
# DPS sampling
res = dps_sampling(model, scheduler, y, forward_model, 1, scale=1, scale_guidance=0)
# Ground truth image x
x = sample['H'].to(device)

plt.figure(figsize=(10, 10/3))
plt.subplot(131)
plt.imshow(util.tensor2uint(y))
plt.axis('off')
plt.subplot(132)
plt.imshow(util.tensor2uint(res))
plt.axis('off')
plt.subplot(133)
plt.imshow(util.tensor2uint(x))
plt.axis('off')
plt.show()

import utils.utils_image as image_utils
image_utils.calculate_psnr(util.tensor2uint(res), util.tensor2uint(sample['H']))
```

```
diffusion_pytorch_model.safetensors not found
env: PYTORCH_ENABLE_MPS_FALLBACK=1
```

```
Loading pipeline components...: 100%                                    2/2 [00:00<00:00, 2.07it/s]
```

```
T t eta_t alpha_prod_t beta_prod_t current_beta_t
100 0 tensor(0.0001) tensor(0.9999) tensor(0.0001) tensor(0.0001)
100 10 tensor(0.0021) tensor(0.9978) tensor(0.0022) tensor(0.0021)
100 20 tensor(0.0041) tensor(0.9937) tensor(0.0063) tensor(0.0041)
100 30 tensor(0.0061) tensor(0.9877) tensor(0.0123) tensor(0.0061)
100 40 tensor(0.0081) tensor(0.9798) tensor(0.0202) tensor(0.0080)
100 50 tensor(0.0101) tensor(0.9700) tensor(0.0300) tensor(0.0100)
100 60 tensor(0.0121) tensor(0.9583) tensor(0.0417) tensor(0.0120)
100 70 tensor(0.0141) tensor(0.9449) tensor(0.0551) tensor(0.0140)
100 80 tensor(0.0161) tensor(0.9299) tensor(0.0701) tensor(0.0159)
100 90 tensor(0.0180) tensor(0.9133) tensor(0.0867) tensor(0.0179)
100%|██████████| 100/100 [00:31<00:00,  3.20it/s]
```

```
6.204011524453935
```

Adaptive η_t

Now we are ready to run the DPS algorithm with the theoretical (adaptive) values of η_t.

First have a look at the scheduler to see how η_t varies with t. `print_scheduler(scheduler, T, 10)` Then set the value of the scale paraameter accordingly.

For best results you should use T=1000 steps. (about 5 minutes)

For faster results you can use T=100 steps. (about 30 seconds)

---

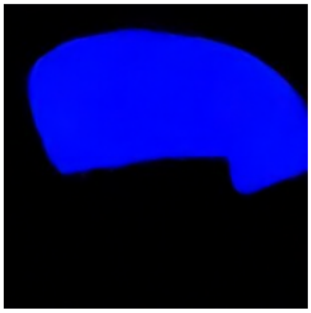
we can clearly see that the value of η_t starts very small and gradually incresed at t increases. this means that more weight is put on the guidance term as the noise level decreases

```
#this is for theoric etat t et ca marche bcp mieux avec pgdm

# torch.roll is not supported for MPS device, so we need to enable CPU fallback
%env PYTORCH_ENABLE_MPS_FALLBACK=1

#scheduler = DDPMScheduler.from_pretrained(model_name)
#model = UNet2DModel.from_pretrained(model_name).to(device)

ddpm = DDPMPipeline.from_pretrained(model_name).to(device)
scheduler = ddpm.scheduler
model = ddpm.unet

T = 100
scheduler.set_timesteps(T)
print_scheduler(scheduler, T, 10)


# Forward model (cuda GPU implementation)
# forward_model = lambda x: agem.fft_blur(x, sample['kernel'].to(device))
# If you are using an MPS gpu use the following forward_model instead
# CPU fallback implementation (no MPS support for torch.roll, fft2, Complex Float, etc.)
forward_model_cpu = lambda x: agem.fft_blur(x, sample['kernel'])
forward_model = lambda x: forward_model_cpu(x.to('cpu')).to(device)
# Degraded image y = A x + noise
y = sample['L'].to(device)
# DPS sampling
res = dps_sampling(model, scheduler, y, forward_model, 1, scale=100, scale_guidance=1)
# Ground truth image x
x = sample['H'].to(device)

plt.figure(figsize=(10, 10/3))
plt.subplot(131)
plt.imshow(util.tensor2uint(y))
plt.axis('off')
plt.subplot(132)
plt.imshow(util.tensor2uint(res))
plt.axis('off')
plt.subplot(133)
plt.imshow(util.tensor2uint(x))
plt.axis('off')
plt.show()

import utils.utils_image as image_utils
image_utils.calculate_psnr(util.tensor2uint(res), util.tensor2uint(sample['H']))
```

```
diffusion_pytorch_model.safetensors not found
env: PYTORCH_ENABLE_MPS_FALLBACK=1
```

Loading pipeline components...: 100% 2/2 [00:00<00:00, 1.49it/s]

```
T t eta_t alpha_prod_t beta_prod_t current_beta_t
100 0 tensor(0.0001) tensor(0.9999) tensor(0.0001) tensor(0.0001)
100 10 tensor(0.0021) tensor(0.9978) tensor(0.0022) tensor(0.0021)
100 20 tensor(0.0041) tensor(0.9937) tensor(0.0063) tensor(0.0041)
100 30 tensor(0.0061) tensor(0.9877) tensor(0.0123) tensor(0.0061)
100 40 tensor(0.0081) tensor(0.9798) tensor(0.0202) tensor(0.0080)
100 50 tensor(0.0101) tensor(0.9700) tensor(0.0300) tensor(0.0100)
100 60 tensor(0.0121) tensor(0.9583) tensor(0.0417) tensor(0.0120)
100 70 tensor(0.0141) tensor(0.9449) tensor(0.0551) tensor(0.0140)
100 80 tensor(0.0161) tensor(0.9299) tensor(0.0701) tensor(0.0159)
100 90 tensor(0.0180) tensor(0.9133) tensor(0.0867) tensor(0.0179)
100%|██████████| 100/100 [00:30<00:00,  3.24it/s]
```


```
4.9739367169556274
```

2) Pseudoinverse-Guided Diffusion Models (ΠGDM))

In ΠGDM, $p(x_0|x_t)$ is approximated with a Gaussian distribution. We have $p(x_0|x_t) \sim \mathcal{N}(\widehat{x_0}, r_t^2)$.

Now since both $p(x_0|x_t)$ and $p(y|x_0)$ are Gaussian, we can compute the posterior $p(y|x_t)$ in closed form:

$$p(y|x_t) \sim \mathcal{N}(H\widehat{x_0}, r_t^2 HH^T + \sigma^2 Id)$$

which leads to:

$$\nabla_{x_t} \log p(y|x_t) = \left(\frac{\partial \widehat{x_0}(x_t)}{\partial x_t}\right)^T H^T (r_t^2 HH^T + \sigma^2 Id)^{-1}(y - H\widehat{x_0})$$

Autograd is able to do most of the job, but we still need to invert $(r_t^2 HH^T + \sigma^2 Id)$. This computation can be expensive, unless H has some special structure. For deblurring, H is a convolution operator, so we can use the FFT to compute the inverse efficiently.

ΠGDM "guidance" is more efficient that DPS, which means we do not need as much iterations as DPS (between 50 and 100). However, the cost of a strong guidance comes with limitations on the complexity of the forward operator H.

Reference:

- ΠGDM: https://openreview.net/forum?id=9_gsMA8MRKQ

```
# nothing to program jus tot see results
%env PYTORCH_ENABLE_MPS_FALLBACK=1
import torch.fft as fft
```