CS726 Programming Assignment – 2 Report

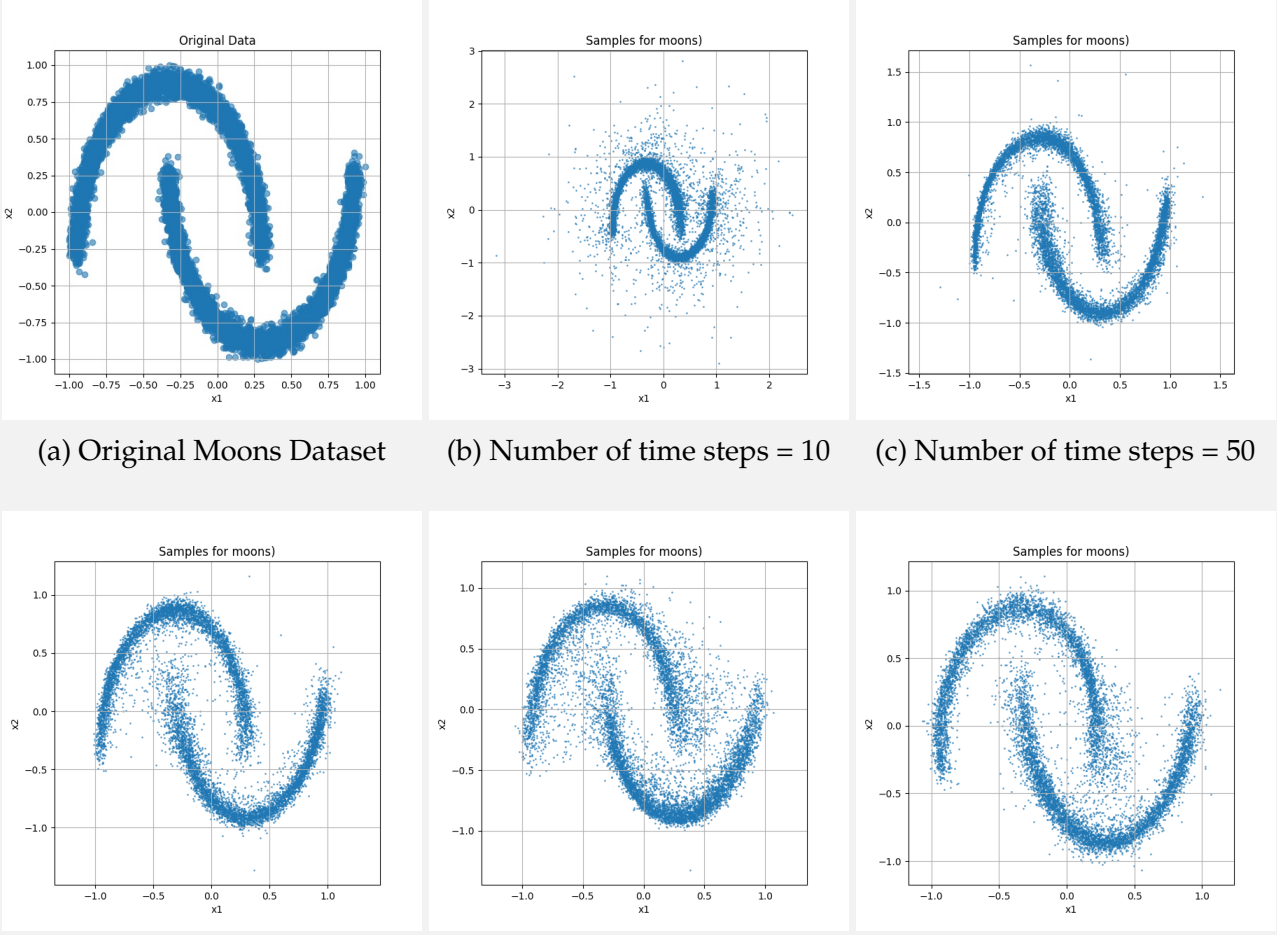
Saksham Rathi (22B1003)
Sharvanee Sonawane (22B0943)
Deeksha Dhiwakar (22B0988)
Department of Computer Science,
Indian Institute of Technology Bombay

Denoising Diffusion Probabilistic Models

Here are the results of unconditional DDPMs on various datasets (with respect to the number of time steps). We had fixed all other parameters (the best settings observed):

- lbeta=0.0001
- ubeta=0.02
- lr=0.0001 (so that training loss decreases across epochs)
- n_samples=10000
- n_dim=2 (for helix it is 3)
- batch_size=128 (to avoid CUDA memory errors and produce optimal results)
- epochs=40

Moons



(a) Original Moons Dataset (b) Number of time steps = 10 (c) Number of time steps = 50

(d) Number of time steps = 100 (e) Number of time steps = 150 (f) Number of time steps = 200

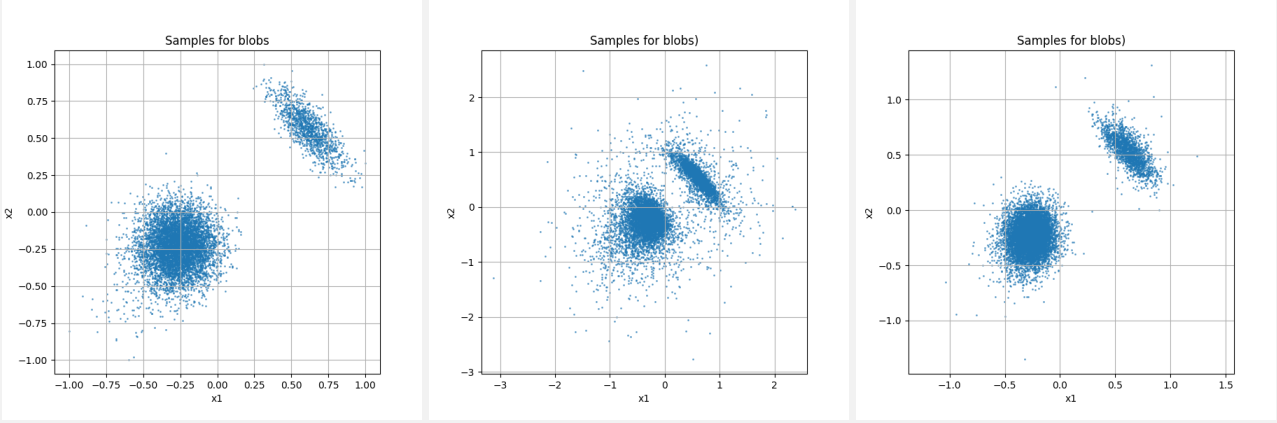
Figure 1: Moons Dataset

Here are the NLL values:

- $T = 10$: 1.048
- $T = 50$: 0.9599
- $T = 100$: 0.9519
- $T = 150$: 0.9218
- $T = 200$: 0.9321

As, we can see from both NLL values and the images, $T = 150$ performed the best.

Blobs



(a) Original Blobs Dataset (b) Number of time steps = 10 (c) Number of time steps = 50

(d) Number of time steps = 100 (e) Number of time steps = 150 (f) Number of time steps = 200

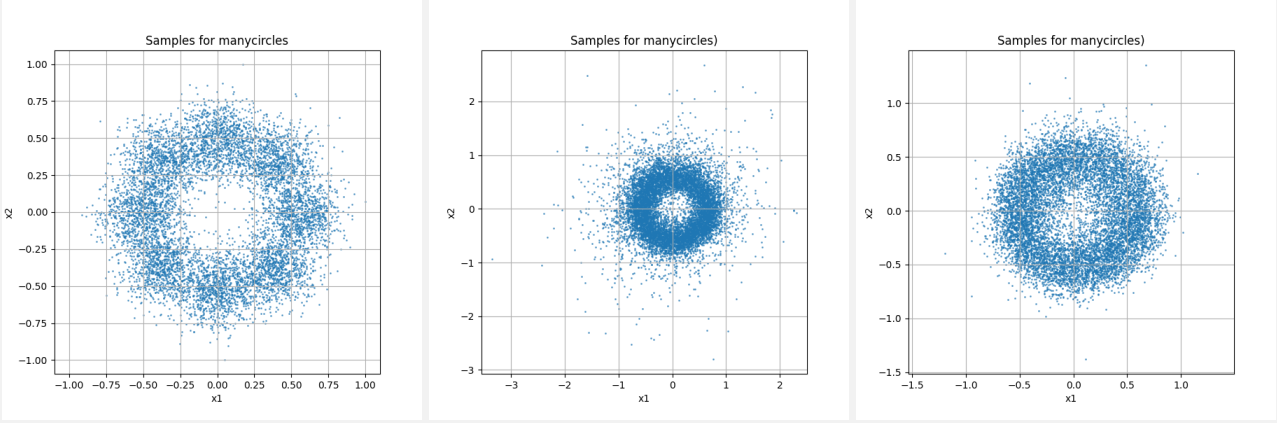
Figure 2: Blobs Dataset

Here are the NLL values:

- $T = 10$: 0.37
- $T = 50$: 0.0152
- $T = 100$: 0.0232
- $T = 150$: -0.0223
- $T = 200$: 0.0045

As, we can see from both NLL values and the images, $T = 150$ performed the best. Moreover, there is a sudden decrease in NLL from 10 to 50, which shows the significant impact of increasing the number of time steps.

Many-Circles



(a) Original ManyCircles Dataset (b) Number of time steps = 10 (c) Number of time steps = 50

(d) Number of time steps = 100 (e) Number of time steps = 150 (f) Number of time steps = 200

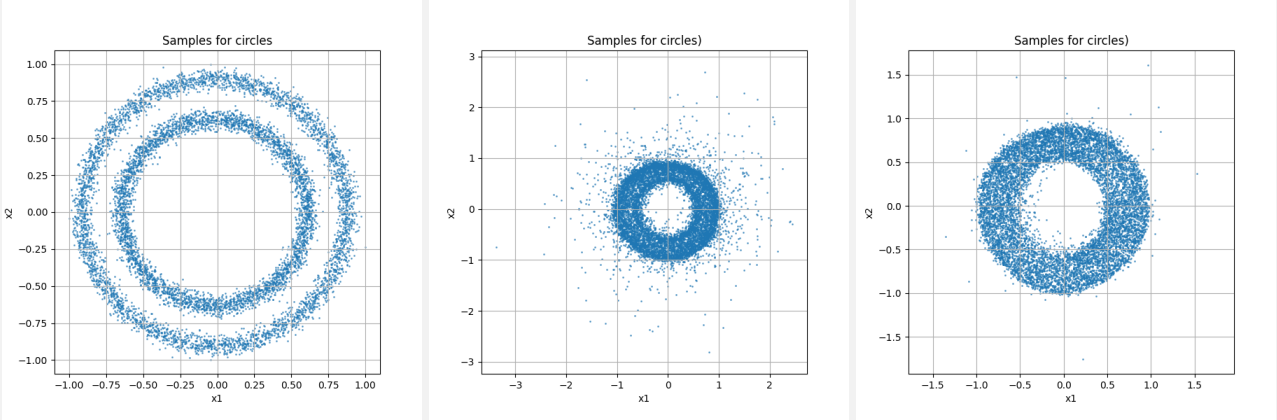
Figure 3: Many Circles Dataset

Here are the NLL values:

- $T = 10$: 0.75
- $T = 50$: 0.548
- $T = 100$: 0.545
- $T = 150$: 0.558
- $T = 200$: 0.522

As, we can see from both NLL values and the images, $T = 200$ performed the best.

Circles



(a) Original Circles Dataset (b) Number of time steps = 10 (c) Number of time steps = 50

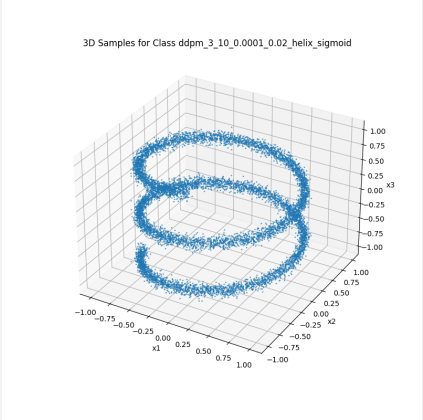
(d) Number of time steps = 100 (e) Number of time steps = 150 (f) Number of time steps = 200

Figure 4: Circles Dataset

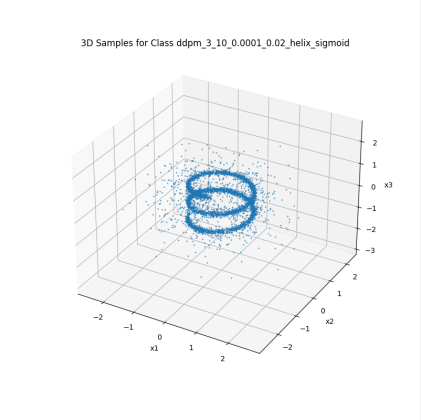
Here are the NLL values:

- $T = 10$: 1.081
- $T = 50$: 0.991
- $T = 100$: 0.9869
- $T = 150$: 1.004
- $T = 200$: 0.992

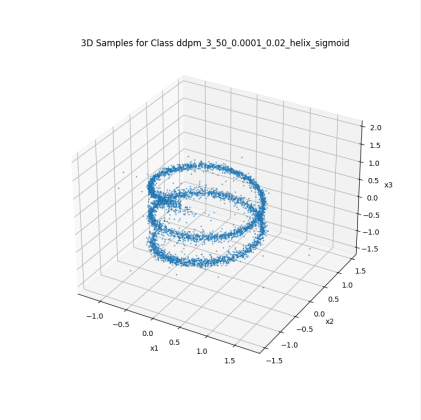
Helix



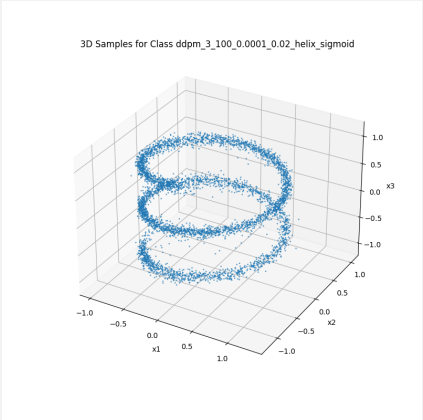
(a) Original Helix Dataset



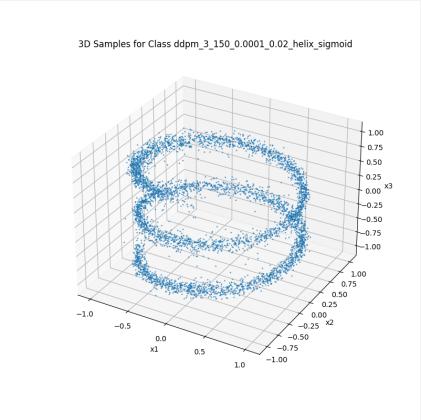
(b) Number of time steps = 10



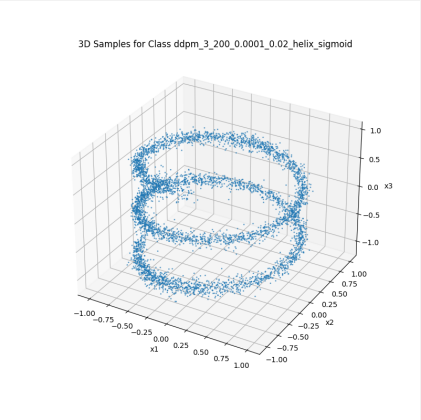
(c) Number of time steps = 50



(d) Number of time steps = 100



(e) Number of time steps = 150



(f) Number of time steps = 200

Figure 5: Helix Dataset

Here are the NLL values:

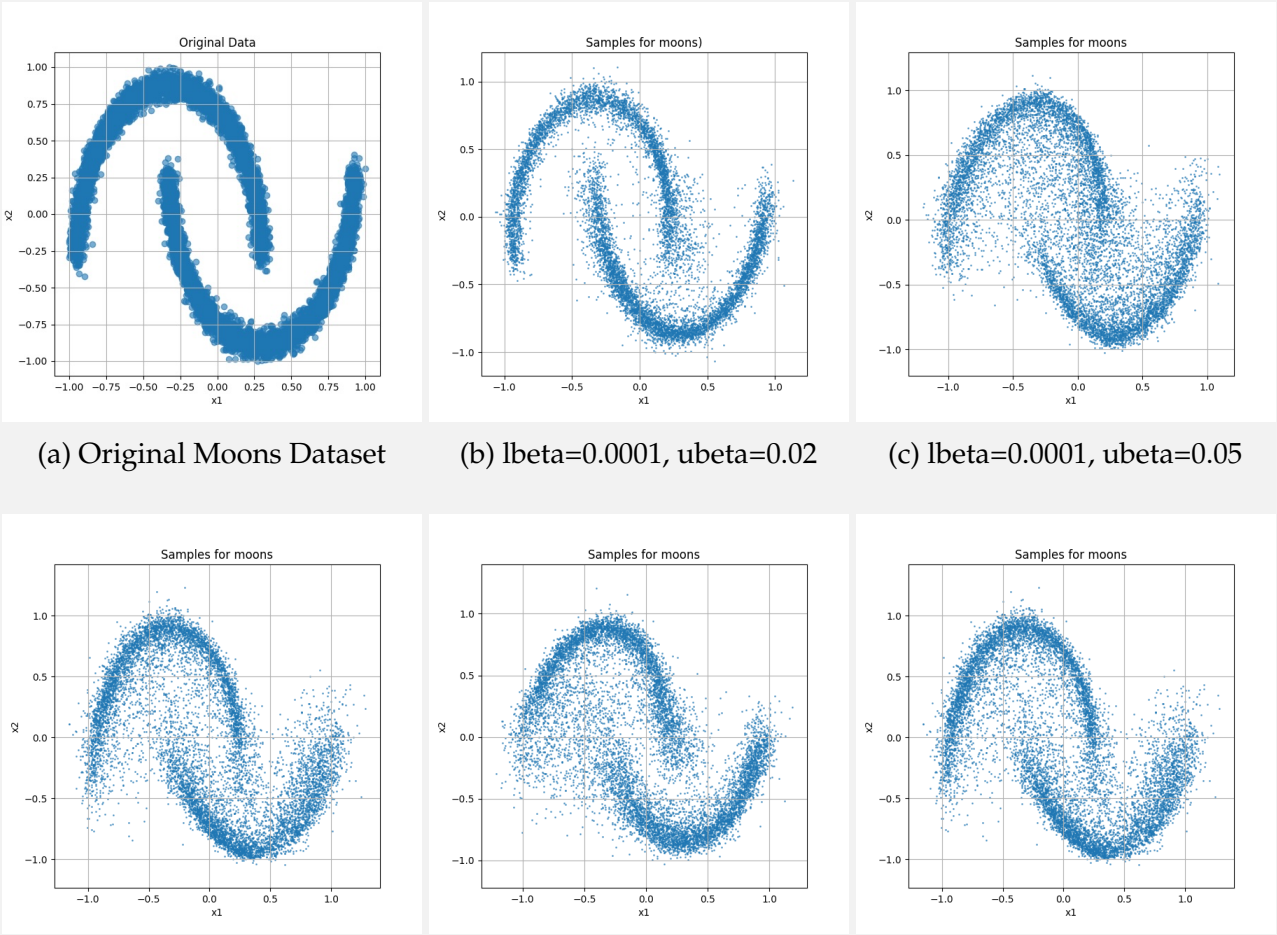
- $T = 10$: 1.6179
- $T = 50$: 1.514
- $T = 100$: 1.5198
- $T = 150$: 1.528
- $T = 200$: 1.528

As we can see from the images (and the NLL values), 50 performs the best.

Noise Schedule Settings

We trained the DDPM for various combinations of ubeta and lbeta values across all the datasets, and selected the best one using NLL value comparison. Here are the NLL values for the moons and blobs datasets:

Moons



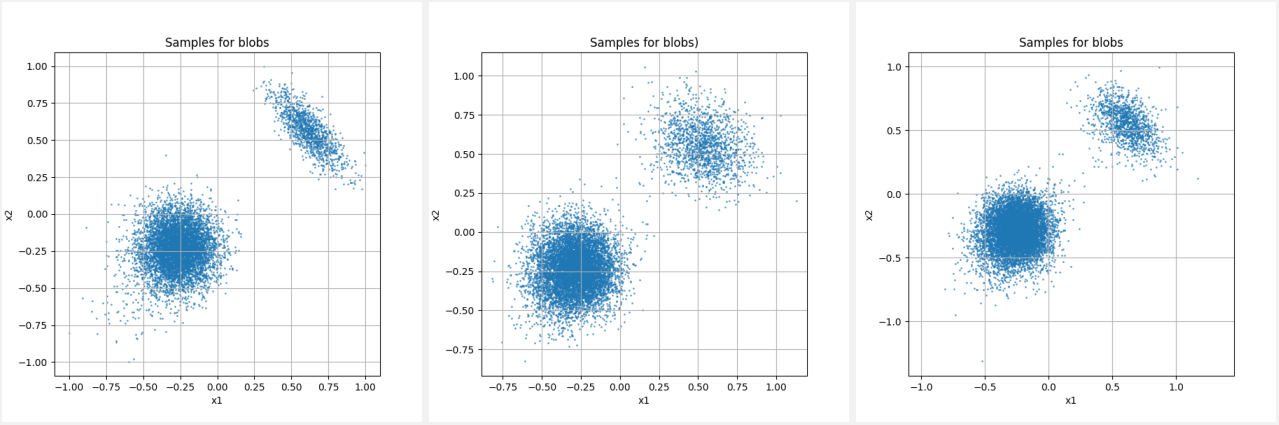
(a) Original Moons Dataset (b) lbeta=0.0001, ubeta=0.02 (c) lbeta=0.0001, ubeta=0.05

(d) lbeta=0.0005, ubeta=0.02 (e) lbeta=0.0005, ubeta=0.05 (f) lbeta=0.001, ubeta=0.05

Figure 6: Moons Dataset

- lbeta = 0.0001, ubeta = 0.02: 0.9184
- lbeta = 0.0001, ubeta = 0.05: 0.9223
- lbeta = 0.0005, ubeta = 0.02: 0.9321
- lbeta = 0.0005, ubeta = 0.05: 0.9265
- lbeta = 0.001, ubeta = 0.05: 0.9498

Blobs



(a) Original Blobs Dataset (b) lbeta=0.0001, ubeta=0.02 (c) lbeta=0.0001, ubeta=0.05

(d) lbeta=0.0005, ubeta=0.02 (e) lbeta=0.0005, ubeta=0.05 (f) lbeta=0.001, ubeta=0.05

Figure 7: Blobs Dataset

- lbeta = 0.0001, ubeta = 0.02: -0.0145
- lbeta = 0.0001, ubeta = 0.05: -0.0102
- lbeta = 0.0005, ubeta = 0.02: -0.0099
- lbeta = 0.0005, ubeta = 0.05: -0.0104
- lbeta = 0.001, ubeta = 0.05: -0.0098

We observe similar NLL trends for the other datasets as well, leading us to conclude that lbeta=0.0001 and ubeta=0.02 are the best hyperparameters for the linear noise schedule.

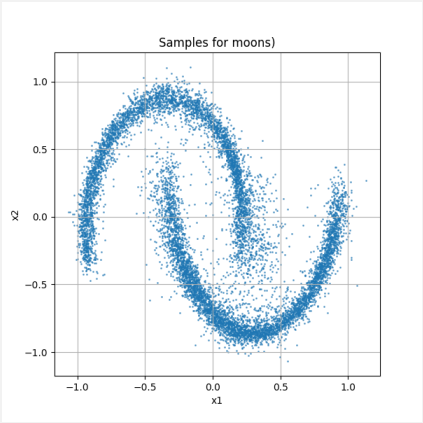
Comparison with Cosine and Sigmoid Noise Schedules

Along with the linear noise schedule, we have studied the effect of cosine and sigmoid noise schedules for the DDP model. The results of the comparison are as shown below. In each of the cases we have set the other hyperparameters as follows:

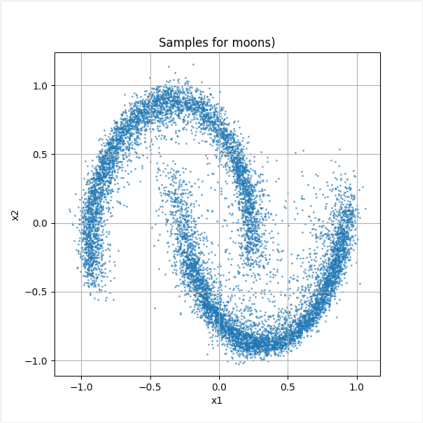
- Number of time steps = 200
- lbeta=0.0001
- ubeta=0.02

- lr=0.0001
- n_samples=10000
- n_dim=2
- batch_size=128
- epochs=40

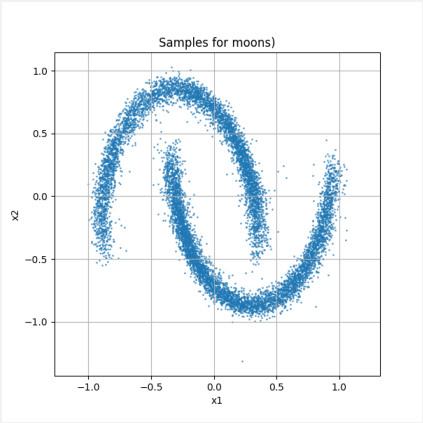
Moons



(a) Linear Noise Schedule
NLL=0.932



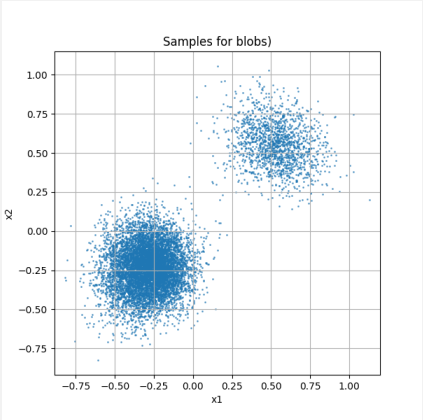
(b) Cosine Noise Schedule
NLL=0.949



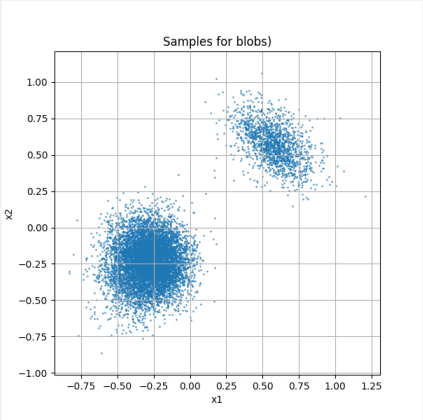
(c) Sigmoid Noise Schedule
NLL=0.928

In this case, we observe that the sigmoid noise schedule is the best out of the three.

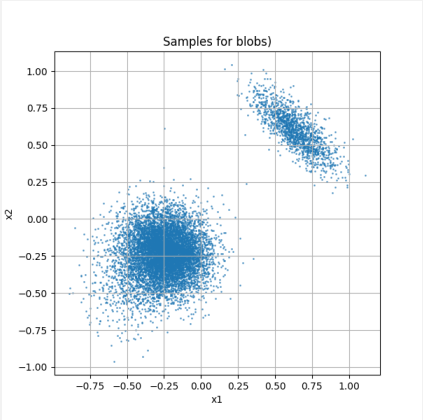
Blobs



(a) Linear Noise Schedule
NLL=0.0045



(b) Cosine Noise Schedule
NLL=0.0043



(c) Sigmoid Noise Schedule
NLL=0.0066

In this case, we observe that the cosine noise schedule is the best out of the three.

Albatross

The results of the comparison between the use of linear, cosine and sigmoid schedule for the albatross dataset are as shown below.

1. Number of time steps = 200, lbeta=0.0001, ubeta=0.02, lr=0.0001, n_samples=10000, n_dim=64, batch_size=128, epochs=40

- NLL for linear schedule = -4.27142733285642112
- NLL for cosine schedule = -5.045544147491455
- NLL for sigmoid schedule = -4.5471272468566895

2. Number of time steps = 200, lbeta=0.0001, ubeta=0.05, lr=0.0001, n_samples=10000, n_dim=64, batch_size=128, epochs=40
   - NLL for linear schedule = -5.325654983520508
   - NLL for cosine schedule = -5.242018222808838
   - NLL for sigmoid schedule = -4.6837568283081055
3. Number of time steps = 200, lbeta=0.001, ubeta=0.05, lr=0.0001, n_samples=10000, n_dim=64, batch_size=128, epochs=40
   - NLL for linear schedule = -5.0962653160095215
   - NLL for cosine schedule = -5.220670223236084
   - NLL for sigmoid schedule = -5.408768177032471
4. Number of time steps = 200, lbeta=0.001, ubeta=0.1, lr=0.0001, n_samples=10000, n_dim=64, batch_size=128, epochs=40
   - NLL for linear schedule = -4.390678405761719
   - NLL for cosine schedule = -4.833913326263428
   - NLL for sigmoid schedule = -4.660123825073242
5. Number of time steps = 200, lbeta=0.01, ubeta=0.05, lr=0.0001, n_samples=10000, n_dim=64, batch_size=128, epochs=40
   - NLL for linear schedule = -4.375370502471924
   - NLL for cosine schedule = NLL: -4.554390907287598
   - NLL for sigmoid schedule = -4.550512790679932

From the results, we observe that the cosine schedule generally achieves the lowest NLL, indicating better model performance in most cases. However, at higher values of `lbeta` and `ubeta`, the sigmoid schedule performs slightly better, as seen when `lbeta` = 0.001 and `ubeta` = 0.05, where sigmoid achieves the lowest NLL (-5.4088). The linear schedule consistently underperforms compared to the cosine and sigmoid schedules.

We train a model on the Albatross dataset using our best-performing hyperparameter configuration. The training is performed using the following parameters:

- **Number of time steps:** 200
- **Lower bound of beta (lbeta):** 0.001
- **Upper bound of beta (ubeta):** 0.05
- **Learning rate (lr):** 0.0001
- **Number of samples (n_samples):** 10,000
- **Dimensionality (n_dim):** 64
- **Batch size:** 128
- **Epochs:** 40
- **Noise schedule:** Sigmoid (as it provided the best NLL score)

Model File and Sampling Process The trained model is saved as `exps/ddpm_64_200_0.001_0.05_albatross_sigmoid/model.pth`. Using this model, we generate samples following the given deterministic process:

- We initialize x_T using `data/albatross_prior_samples.npy`, which contains 32,561 vectors sampled from $\mathcal{N}(0, I_d)$.
- In step 4 of Algorithm 2 (from [2]), we set $z = 0$, ensuring deterministic execution.

The generated samples are saved in `albatross_samples.npy`.

Reproducibility To verify and reproduce our results, we provide a script `reproduce.py`, which:

1. Loads the trained model (`model.pth`).
2. Regenerates the samples using the deterministic process.
3. Saves the reproduced samples as `albatross_samples_reproduce.npy`.

Classifier-Free Guidance

Difference between Guided Sampling and Conditional Sampling

In conditional sampling, we model $p(x|y)$ directly, where y is a conditioning variable (like a class label). During generation, we sample from this conditional distribution to get samples that match the condition.

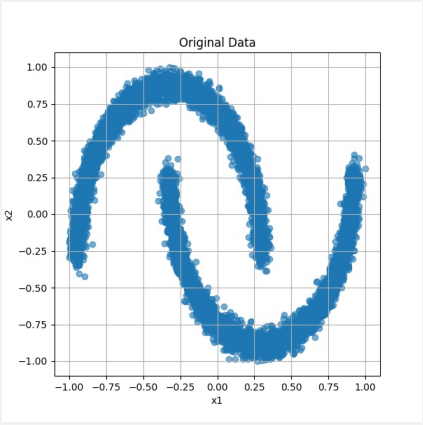
In guided sampling (specifically classifier-free guidance), we train two models: one conditional $p(x|y)$ and one unconditional $p(x)$. During sampling, we interpolate between them with a guidance scale w:

$$\epsilon_\theta(x_t, t, y) = (1 + w) * \epsilon_\theta(x_t, t, y) - w * \epsilon_\theta(x_t, t)$$

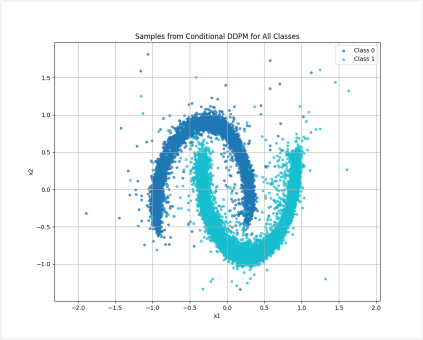
Firstly, guided sampling is more expensive in terms of computation required to train and sample points (almost double, since we are training two models, and using both of them to sample points). However, this guidance increases the impact of the conditioning information and can produce higher quality samples that better match the condition, but with a potential loss of diversity.

Effect of guidance scale

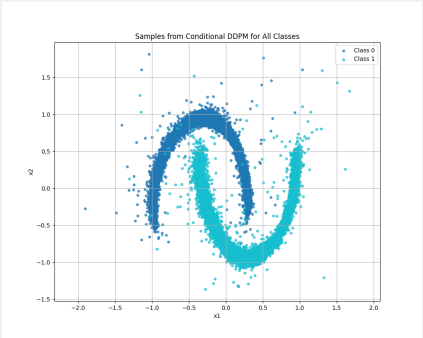
We sampled points using CFG on a variety of guidance scale values, here are the images from the moon dataset:



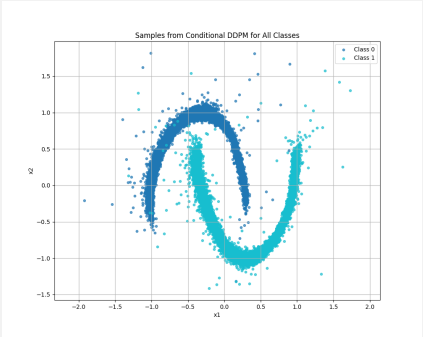
(a) Original Moons Dataset



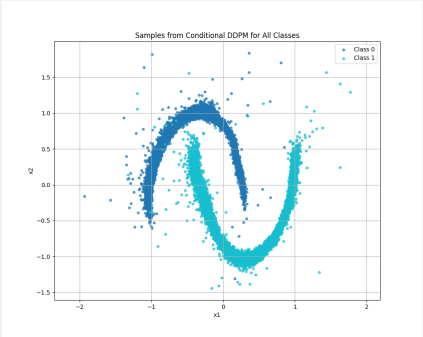
(b) Guidance Scale = 0 (Equivalent to Conditional Sampling)



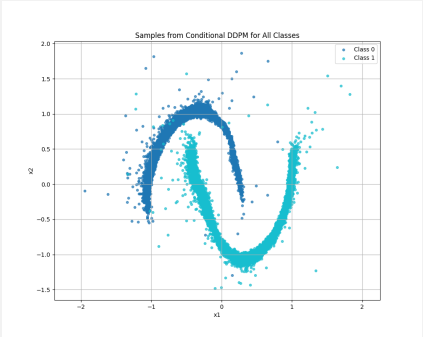
(c) Guidance Scale = 0.2



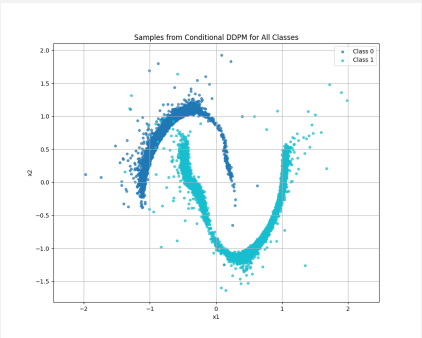
(d) Guidance Scale = 0.5



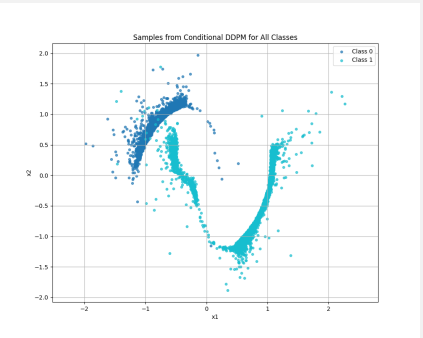
(e) Guidance Scale = 0.7



(f) Guidance Scale = 1.0



(g) Guidance Scale = 2.0



(h) Guidance Scale = 4.0

Figure 10: Moons - CFG

Here are the NLL values for the two class labels:

Guidance Scale	NLL for Class Label 0	NLL for Class Label 1
0.0	0.53	0.58
0.2	0.51	0.58
0.5	0.50	0.59
0.7	0.50	0.60
1.0	0.51	0.62
2.0	0.57	0.66
4.0	0.65	0.71

Table 1: NLL Values for the two class labels of the Moons Dataset

As, it is evident from the NLL values and the images shown above, guidance scales 0.2 and 0.5 perform the best. With higher guidance scales, we typically lose focus on randomly generated data, and rely too much on the training labels provided, thus the NLL values are low, when another sample is chosen from the actual dataset. These observations match with the ones presented in the CFG paper too.

We had also compared the samples generated with a simple classifier model trained, here are the accuracy results across various guidance scale values:

Guidance Scale	Accuracy for Class Label 0	Accuracy for Class Label 1
0.0	0.52	0.49
0.2	0.52	0.50
0.5	0.52	0.51
0.7	0.52	0.52
1.0	0.52	0.52
2.0	0.49	0.55
4.0	0.46	0.61
8.0	0.44	0.67
10.0	0.44	0.68

Table 2: Accuracy Values for the two class labels of the Moons Dataset

As we can see, the accuracy values do not vary much across various guidance scale values. They peak around a scale of 1 (which is as per the results obtained in the paper), similar to NLL results. Higher scale values tend to bias towards one of the class labels as demonstrated in the images too.

Classifier using Conditional DDPM

The `ClassifierDDPM` class implements a classification method using a Diffusion Probabilistic Model (DDPM). The classification is performed by computing the likelihood of each class given the input, using a pre-trained conditional DDPM.

Given an input $x \in \mathbb{R}^d$, we generate a noisy sample x_t at a fixed timestep t using the variance-preserving noise schedule:

$$x_t = \sqrt{\bar{\alpha}_t}x + \sqrt{1-\bar{\alpha}_t}\epsilon, \tag{1}$$

where:

- $\bar{\alpha}_t$ is the cumulative product of noise scheduling parameters,
- $\epsilon \sim \mathcal{N}(0, I)$ is Gaussian noise.

For each class label $c \in 0, 1, \ldots, C-1$, the classifier uses the conditional DDPM model to predict the noise:

$$\hat{\epsilon}_c = \text{model}(x_t, t, c), \tag{2}$$

where $\hat{\epsilon}_c$ is the predicted noise when conditioning on class c.

The classification score for class c is computed as the negative mean squared error between the predicted and actual noise:

$$S_c = -\mathbb{E}\left[|\hat{\epsilon}_c - \epsilon|^2\right]. \tag{3}$$

This score measures how well the model's noise prediction aligns with the actual noise for each class. A lower error implies a higher likelihood of the class being correct.

To convert the classification scores into probabilities, a softmax function is applied:

$$P(c \mid x) = \frac{\exp(S_c)}{\sum_{c'} \exp(S_{c'})}. \tag{4}$$

The predicted class is then obtained as:

$$\hat{c} = \arg\max_{c} P(c \mid x). \tag{5}$$

Let us now compare this classifier with the one trained for the previous part (we will calculate the accuracy of both of these classifiers from the data generated).

Dataset	Trained Classifier Accuracy	ClassifierDDPM Accuracy
Moons	100%	99.19%
Blobs	96.88%	96.56%
Circles	100.00 %	98.25%
Manycircles	85.69%	85.19 %

Table 3: Trained Classifier vs Classifier DDPM

As we can see, classifierDDPM performs equally well as compared to the trained Classifier. The accuracy is lower in case of manycircles, because of larger number of classes, as compared to other datasets.

Reward Guidance

Denoising Diffusion Probabilistic Models (DDPM) generate samples by progressively denoising a noisy latent variable. The reverse process of diffusion is modeled using a conditional Gaussian distribution:

$$q(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_t(x_t), \Sigma_t) \tag{6}$$

where $\mu_t(x_t)$ is the **posterior mean estimate** at step t. In DDPM, this mean is typically approximated using a **pre-trained noise predictor** $\epsilon_\theta(x_t, t)$, which estimates the noise added at timestep t:

$$\mu_t(x_t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}t}} \epsilon_\theta(x_t, t) \right) \tag{7}$$

where:

α_t, β_t are diffusion process parameters,

Soft Value-Based Decoding (SVDD) modifies the denoising process by introducing a **reward function** $R(x_t)$, which biases the sampling towards high-reward samples. The key idea is to adjust the **posterior mean approximation** using a value-based gradient:

$$\tilde{\mu}t(x_t) = \mu_t(x_t) + \lambda R(x_t) \nabla x_t \log p(x_t) \tag{8}$$

where:

$R(x_t)$ is the reward function (e.g., classifier-based reward),

λ (reward scale) controls the influence of the reward function,

$\nabla_{x_t} \log p(x_t)$ is an estimate of the score function (guiding towards high-reward regions).

Effectively, SVDD **shifts the denoising trajectory** toward samples that maximize the reward while still following the probabilistic structure of DDPM.

In practice, $\nabla_{x_t} \log p(x_t)$ is difficult to compute directly. Instead, we approximate it using:

Energy-Based Models (EBMs): Estimate the log-probability of samples using a learned energy function.

Pre-trained Classifiers: If a classifier is available, we use its log-probability gradient as an approximation. (This is the reward function which was used by us.)

SVDD modifies the posterior mean approximation in DDPM by incorporating **reward-based adjustments**. This allows guiding the generative process **without explicit conditioning**, making it useful for sampling high-reward samples in an **unconditional** model.

Dataset	Trained Classifier	ClassifierDDPM	SVDD
Moons	100%	99.19%	64.15%
Blobs	96.88%	96.56%	57.89%
Circles	100.00 %	98.25%	77.62%
Manycircles	85.69%	85.19 %	30.04%

Table 4: Trained Classifier vs Classifier DDPM

As we can see from the accuracy values of the table, SVDD performs quite better as compared to classifierDDPM which was used in the previous part.